2005

VHDL Modeling of an H.264/AVC Video Decoder

Thomas Benjamin Warsaw

VHDL Modeling of an H.264/AVC Video Decoder

by

Thomas Benjamin Warsaw

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Visiting Instructional Faculty, RIT Department of Computer Engineering Dr. Marcin Lukowiak
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August 2005

Approved By:

Marcin Lukowiak

Dr. Marcin Lukowiak
Visiting Instructional Faculty, RIT Department of Computer Engineering
Primary Adviser

Andreas Savakis

Dr. Andreas Savakis
Professor, RIT Department of Computer Engineering

Kenneth Hsu

Dr. Kenneth Hsu
Professor, RIT Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology

Kate Gleason College of Engineering

Title: VHDL Modeling of an H.264/AVC Video Decoder

I, Thomas Benjamin Warsaw, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Thomas Benjamin Warsaw

Thomas Benjamin Warsaw

Date

Dedication

To my parents, for teaching me to love learning,
to my wife, for her loving support,
and to Jesus Christ, for bringing me through.

Acknowledgments

Special thanks to all of my advisors for giving of their time and knowledge; and especially to Dr. Lukowiak for always finding time to meet with me and being more gracious than I deserve. Thanks also to William Batts for working with me on the initial implementation of the transform unit.

Abstract

Transmission and storage of video data has necessitated the development of video compression techniques. One of today's most widely used video compression techniques is the MPEG-2 standard, which is over ten years old. A task force sponsored by the same groups that developed MPEG-2 has recently finished defining a new standard that is meant to replace MPEG-2 for future video compression applications. This standard, H.264/AVC, uses significantly improved compression techniques. It is capable of providing similar picture quality at bit rates of 30-70% less than MPEG-2, depending on the particular video sequence and application [20].

This thesis developed a complete VHDL behavioral model of a video decoder implementing the Baseline Profile of the H.264/AVC standard. The decoder was verified using a testing environment for comparison with reference software results. Development of a synthesizable hardware description was also shown for two components of the video decoder: the transform unit and the deblocking filter. This demonstrated how a complete video decoder could be developed one module at a time with individual module verification. Analysis was also done to estimate the performance and hardware requirements for a complete implementation on an FPGA device.

Contents

List of Figures

List of Tables

Glossary

C

C++ A widely used object-oriented software programming language.

CABAC Context Adaptive Binary Arithmetic Coding. A highly-efficient entropy coding standard used in the H.264/AVC Main Profile.

CAVLC Context Adaptive Variable Length Coding. An improved, context-adaptive version of VLC used in the H.264/AVC Baseline Profile.

D

DCT Discrete Cosine Transform. A matrix transform commonly used to convert image data from the spatial domain into the frequency domain.

DVD Digital Video Disk (also Digital Versatile Disk). A popular optical disk storage technology used for videos and other applications that require large amounts of storage.

H

H.264/AVC Video coding standard approved in Spring 2003 by both ISO/IEC and ITU-T. Delivers significantly better compression than previous standards such as MPEG-2.

HDTV High definition television. A number of high-quality resolutions standardized for television use. Includes 1080x720 and 1920x1080 resolutions.

I

IDR Instantaneous Data Refresh. A frame that signals the reference picture list that any previous reference frames will no longer be needed.

ISO/IEC International Standards Organization/International Electrotechnical Commision. ISO is an international body responsible for creating and maintaining a wide range of standards. The IEC is the commission specifically responsible for electrical products and components, including MPEG video compression standards.

ITU-T International Telecommunications Union (ITU) Telecommunications Standardization Sector. Responsible for developing worldwide standards for telecommunications technology.

M

MPEG Moving Picture Experts Group. The group within ISO/IEC that is responsible for adopting and defining video compression standards.

MPEG-2 Video coding standard created by the MPEG group; used extensively for cable television broadcasting and DVDs.

N

NAL Network Abstraction Layer. The layer in H.264/AVC that defines how video payloads are stored or transmitted.

P

PSNR Peak Signal-to-Noise Ratio. A measure of the objective quality of an image.

Q

QCIF Quarter-resolution Common Image Format. Defines an image size of 176 pixels wide by 144 pixels high.

R

RAM Random Access Memory. Type of reusable data storage that can be accessed in any order.

RBSP Raw Bit Sequence Payload. The payload containing the actual packet information inside a NAL unit.

V

VCEG Video Coding Experts Group. A group from the ITU-T responsible for adopting and defining video compression standards.

VCL Video coding layer. The layer in the H.264/AVC standard that contains actual video information.

VHDL Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (HDL). A popular language used for modeling and describing hardware.

VHS Video Home System. The tape format used in most consumer Video Cassette Recorders (VCRs).

Chapter 1

Introduction

1.1 Background

Video technology uses a rapid sequence of pictures to portray visual information to the human eye. Representing this information digitally can require a vast amount of data. A color picture containing 352x288 pixels requires 304,128 bytes with full depth of color information (24 bits per pixel) and 152,064 bytes for reduced color depth (12 bits per pixel). When considering video signals with 30 frames per second, the amount of data is increased significantly, *e.g.* VHS video quality, 352x288 pixels per frame, 12 bits per pixel, requires 36.5 Mbps; HDTV quality, 1920x1080 pixels per frame, 12 bits per pixel, requires a massive 1493.0 Mbps.

Transmission and storage of this huge amount of data calls for an effective means of compression. This can be achieved in two main ways: by getting rid of statistical redundancy in the video data and by removing high frequency details and color depth that the human eye is less sensitive toward. By doing this, the perceived quality can be maintained while minimizing the amount of data needed.

With the continual development of video applications in recent years, there has been a strong push for compression standards that deliver better picture quality with a smaller bit rate. The H.264/MPEG-4 AVC (Advanced Video Coding) standard was developed to improve on current compression standards like MPEG-1, -2, and -4.

As with MPEG-2 video compression, H.264/AVC is based on block transforms and

motion compensated predictive coding, but uses improved coding techniques including:

- Multiple reference frames
- Intra-frame prediction
- 1/4 pixel precision motion compensation
- More block sizes for motion compensation
- A 4x4 integer transform that approximates the DCT with a much simpler algorithm
- In-loop deblocking filter to remove blocking artifacts and increase final picture quality
- Improved entropy coding with CABAC and CAVLC
- Error resilience tools for maintaining video quality in error-prone broadcasting

These coding techniques provide better video compression than previous standards. Independent lab tests show that H.264/AVC can maintain video quality while offering a 30 to 70 percent improvement in bit rate reduction over MPEG-2 [20], making it much more effective for delivering high-quality video over cable, satellite, and telecom networks. However, this improved compression requires significantly more processing power than previous video standards [21]. Because of this increased complexity, the widespread adoption of H.264/AVC may be limited unless efficient and cost-effective hardware implementations are developed for real-time encoding and decoding of high resolution video [6].

Hardware designers trying to develop hardware implementations of H.264/AVC encoders or decoders face the challenge of verifying their designs. While reference software is available to demonstrate the expected results of the encoding or decoding process, verifying individual stages of a hardware design is difficult. Designers have to develop the complete encoder or decoder in order to verify its operation. Not only does this make it difficult to identify and correct errors in a new hardware design, but it also prevents new designers from focusing on the development of hardware for a single stage.

1.2 Thesis Objective

This thesis provides an initial implementation of an H.264/AVC video decoder that can be used as a basis for future research into H.264/AVC and other video codecs. The H.264/AVC video decoder was implemented with the functionality of the H.264/AVC Baseline Profile, and it was designed to be easily expanded in the future to include the features of the other H.264/AVC profiles. Figure 1.1 shows the main components of the decoder, including the stream parser, the inverse quantizer and inverse transform unit, the inter and intra prediction units, the deblocking filter, and the frame buffer.

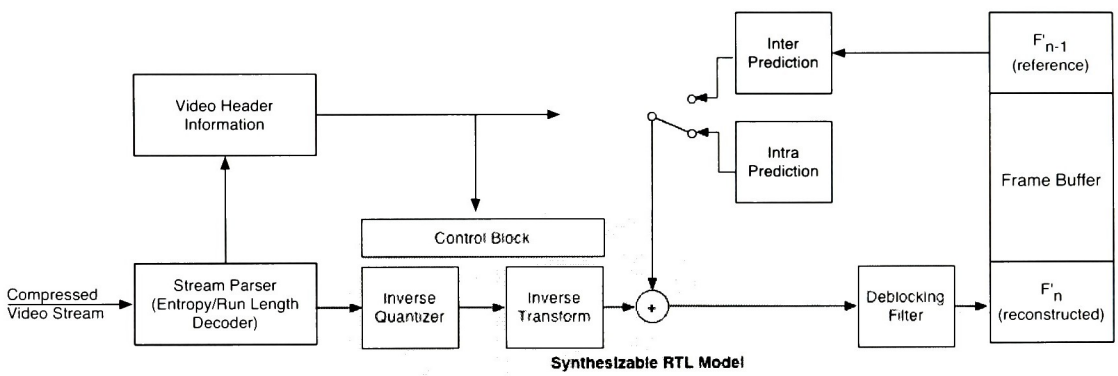

Figure 1.1: Video decoder partitioning

The entire decoder was modeled behaviorally using the VHDL hardware description language. This provided a basis for testing the overall operation of the decoder and will allow future hardware implementations of each module to be tested against the behavioral models. In addition to the complete behavioral model, synthesizable descriptions were created for the inverse quantizer, inverse transform, and deblocking filter modules. The shaded area in Figure 1.1 shows which portion of the decoder was described using synthesizable VHDL.

The synthesizable decoder components were created to provide an estimate of their hardware complexity and performance. The results were also used to analyze the relative performance, power consumption, and cost of implementing an entire H.264/AVC video

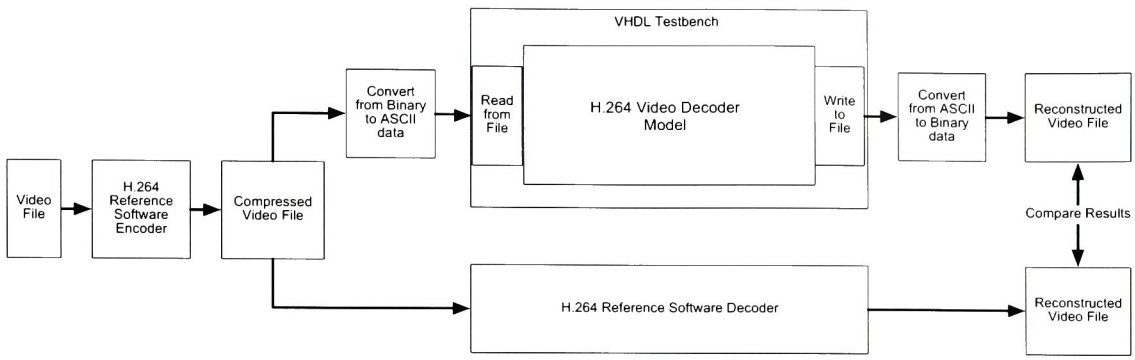

Figure 1.2: Decoder test setup

decoder using various field-programmable gate array (FPGA) devices.

The VHDL model was verified using a reference H.264/AVC codec written in C++ (obtained from [2]). The software reference encoder was used to compress two test video sequences. The encoded video files were then decoded using both the reference software decoder and the behavioral model. A VHDL testbench handled reading an encoded file, sending it through the decoder, and writing out the final video file. The reconstructed video files were then compared to verify that the VHDL model produced the same results as the reference software. A typical test setup is shown in Figure 1.2.

1.3 Thesis Chapter Overview

This thesis starts with an overview of video compression techniques in Chapter 2. An overview of basic video compression techniques is presented along with a look at the changes implemented in H.264/AVC. Chapter 3 then provides a more in-depth look at the decoding algorithms defined by the H.264/AVC standard. Chapter 4 discusses the specific decoder implementation created for this thesis, including details on the behavioral and synthesizable VHDL descriptions.

The test environment created to test the decoder implementation is presented in Chapter 5. Test results are then shown in Chapter 6, along with the results of creating a hardware implementation of the synthesizable parts of the decoder. Finally, Chapter 7 concludes

the thesis with a discussion of the results and a look at future work that could be done on the decoder. It also proposes a hardware architecture for future implementations of the H.264/AVC video decoder.

Chapter 2

Video Compression Techniques

This chapter briefly discusses video compression history and provides an overview of the techniques used by MPEG-2 and the improvements made in H.264/AVC.

2.1 Video Compression History

Over the last two decades, two main groups have been responsible for standardizing video compression techniques: the Video Coding Experts Group (VCEG) of ITU-T and the Moving Picture Experts Group (MPEG) of ISO/IEC. Figure 2.1 shows the video standards produced by these groups, targeting a wide range of applications from video teleconferencing to TV broadcasting and DVDs.

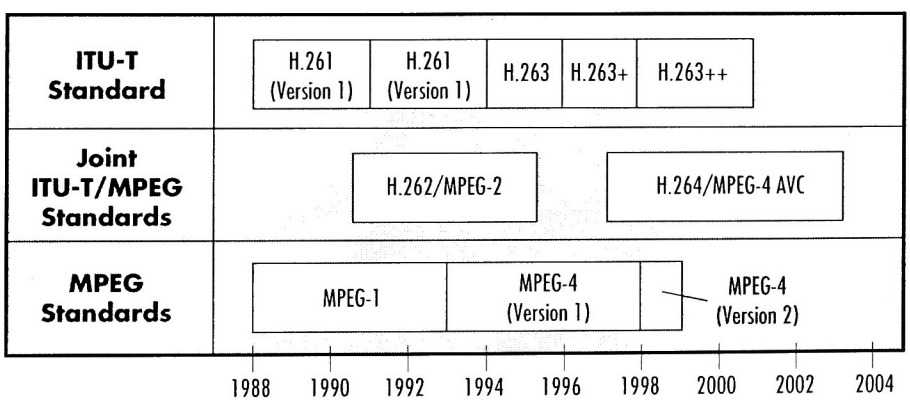

Figure 2.1: History of video standards [1]

The ITU-T is responsible for the H- series of video standards, which especially target video conferencing applications. Their most recent video conferencing standard, H.263, has undergone two major revisions to produce H.263++ (also called H.263 High Latency Profile (HLP)). The MPEG series of video standards have especially targeted high-end video applications. MPEG-2 is currently used for DVDs and broadcast television. MPEG-4 ASP (Advanced Simple Profile), also called MPEG-4 Version 1, was developed primarily for Internet video streaming applications.

MPEG-2 (also known as H.262) and H.264/AVC are the only two video standards ever to be developed jointly by ITU-T and ISO/IEC. H.264/AVC, which was approved in May 2003, represents the results of five years of effort to develop a standard with double the compression rate of MPEG-2. H.264/AVC addresses the full range of video applications, from low-bandwidth wireless uses, low- and high-definition television, video streaming over the Internet, high-quality DVD content, and extremely high-quality video for use in movie theaters [21]. Now that it has been approved, it is expected to eventually replace MPEG-2 for international mass market acceptance [1].

The video compression techniques used by H.264/AVC follow the basic encoding and decoding steps used by MPEG-2. Understanding these basic techniques can help to gain an understanding of how H.264/AVC operates and how it is able to achieve higher compression rates.

2.2 MPEG-2 Video Compression Overview

MPEG-2 is currently the most widely used standard for video compression. DVDs and cable television transmissions use MPEG-2 compression to enable video to be stored or transmitted using limited bandwidth.

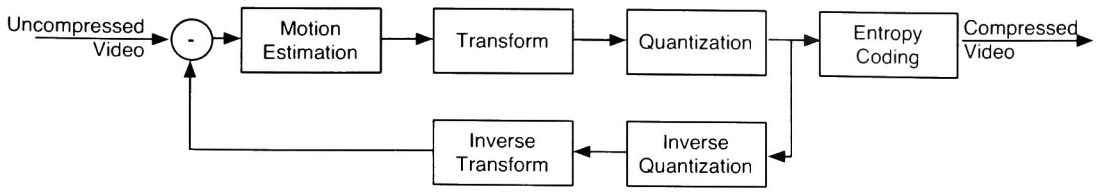

Figure 2.2: MPEG-2 video encoding flowchart

2.2.1 MPEG-2 Encoding

The basic steps of video compression in MPEG-2 are shown in the flowchart of Figure 2.2. The processing stages include:

- Motion Estimation
- Transform (and Inverse Transform)
- Quantization (and Inverse Quantization)
- Entropy Coding

The uncompressed video is sent through the motion estimation, transform, and quantization stages. This data is sent through a feedback path that reproduces the video decoding that would be done by a decoder. This pseudo decoder output is used to compute the difference between the actual incoming video and the estimated/transformed/quantized video. These differences, called residual video data, are what is actually encoded into the compressed video stream, along with the settings that were used by the motion estimation, transform, and quantization stages to estimate the original video frame. These estimation settings and the residual data are then sent through the entropy coding stage to remove statistical redundancies in the data stream and to produce the final compressed video stream.

Each stage of the encoder processes one 16x16 pixel macroblock of a video data at a time. This allows the processing of a complete video frame to be pipelined for a streamed transmission. Encoding parameters are transmitted as header information for each macroblock and for each collection of macroblocks (called a slice) that make up a video frame.

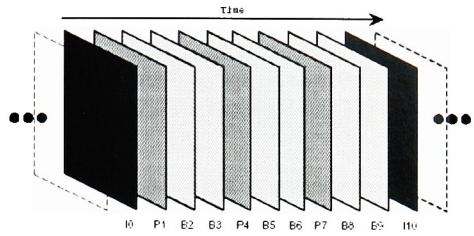

Figure 2.3: Typical Group of Pictures (GOP) in coded order [3]

In addition to this, parameters used for the overall video sequence and for each frame can be transmitted as separate information that can be referred to by each slice header.

Each of the processing states in the encoder is described in more detail below.

Motion Estimation

Since multiple video frames are displayed each second, long sequences of image frames can contain very similar data. Motion estimation compares the sequence of image frames in a video to find temporal redundancies and only encode the changes that occur between frames. These changes are often confined to specific portions of the image where movement is occurring, allowing motion estimation techniques to result in a large decrease in the video stream size [1].

Three different types of picture frames can be encoded by the motion estimation block: I, P, and B, as shown in Figure 2.3. I-frames are coded independently from any other frames. These provide a baseline reference for the other frames to be decoded from. Because they include a full picture frame worth of data, they can only be compressed moderately. P-frames are predictively coded picture frames, encoded with reference to previous I- or P- frames. B-frames (bi-directionally predictive-coded frames) are the most highly compressed type of frame, making reference to both past and future I- or P- frames in the video sequence [3].

Transform

The transform stage transforms the image data from the spatial domain into the frequency domain. MPEG-2 uses a Discrete Cosine Transform (DCT) on each 8x8 block in the image [1]. This transformation reorders the block data according to its frequency, grouping low frequency (more significant) information together.

Quantization

Quantization takes the frequency coefficients produced by the transform stage and removes coefficients for high frequencies. The amount of quantization can be adjusted depending on the desired image quality and compression rate. Removing high frequency coefficients removes image information but maintains most of the perceptual quality (since the human eye cannot distinguish high frequency detail very well) [1].

After the transform and quantization stages, coefficients are in order from lower frequency to higher frequency. Since higher frequency coefficients tend to be zero, this ordering produces a considerable coding improvement in the entropy coding stage. During compression, long sequences of zeros can be represented by a simple bit pattern [1].

Entropy Coding

Entropy coding compresses the final serial data stream by mapping frequently used symbols to actual bit codes. The most frequently occurring symbols are mapped to shorter bit codes, while less frequently occurring symbols are mapped to longer bit codes. This lossless encoding reduces the bandwidth of the final video stream while allowing the data to be completely reconstructed after transmission.

2.2.2 MPEG-2 Decoding

After the video has been transmitted or stored, the original video sequence is reconstructed by reversing the encoding stages. The decoding flow is shown in Figure 2.4. The data

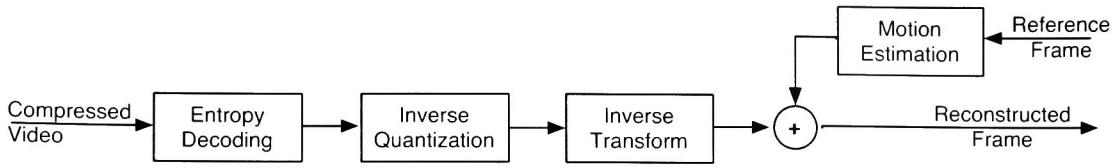

Figure 2.4: MPEG-2 video decoding flowchart

compression is first undone by the entropy decoding stage. The residual video data is then reconstructed by the inverse transform and inverse quantization stages. The motion compensation block generates a prediction based on previously decoded reference frames (if any). This prediction is added to the residual data to get the final decoded video frame.

2.3 H.264/AVC Video Compression

H.264/AVC follows the basic video encoding and decoding steps outlined above, but additional techniques are included that allow H.264/AVC to achieve 30-70% better compression than MPEG-2, as well as substantial perceptual quality improvements [21], [20].

Three different encoding profiles are defined in H.264/AVC. Each profile consists of a particular set of encoding functionality that is required to be supported by any encoder or decoder that implements that profile. The functionality supported by each profile is somewhat overlapping. The Baseline Profile supports intra and inter prediction using I and P frames and entropy coding using Context Adaptive Variable Length Coding (CAVLC). The Main Profile includes support for interlaced video, inter prediction using B frames and weighted prediction, and entropy coding using Context Adaptive Binary Arithmetic Coding (CABAC). The Extended Profile does not support interlaced video or CABAC, but adds modes for efficient switching between coded bitstreams and improved error resilience using data partitioning [15].

Characteristics	MPEG-2	H.264/AVC
Block shapes	Square	Arbitrary
Block sizes	Large (16x16)	Small (4x4)
Motion vector resolution	1/2 pixel	1/4 pixel (1/8 chroma)

Table 2.1: H.264/AVC motion estimation comparison [1]

2.3.1 Changes from MPEG-2

Some of the new compression techniques in H.264/AVC simply remove restrictions placed on encoding in MPEG-2, while other changes are more major. Spatial compression has been improved with the addition of the intra estimation stage, new compression features have been added to the motion estimation stage, and areas such as the image transform and quantization stages have had the compression algorithm completely changed. Some of the notable improvements in H.264/AVC are outlined below.

Motion Estimation

The new standard offers several improvements in how motion estimation can be done. H.264/AVC introduces smaller block sizes, greater flexibility in block shapes, and greater precision in motion vectors. This can result in a much higher temporal compression because of the improved motion prediction that can be accomplished. The changes in how motion estimation can be done over MPEG-2 are highlighted in Table 2.1.

H.264/AVC also introduces the ability to use multiple reference frames for motion estimation. This improves compression by allowing easier description of the following types of video:

- Motion that is periodic in nature
- Translating motion and occlusions
- Alternating camera angles that switch back and forth

Intra Estimation

A new feature added by H.264/AVC is intra estimation. Where motion estimation doesnt work, intra estimation can be used to spatially compress an image. Intra estimation works by extrapolating neighboring pixels into a set of defined directions. This works particularly well on flat backgrounds where the image changes in some consistent way [1]. An example of intra estimation is shown in Figure 2.5. In this example, a 4x4 block is predicted by horizontally extrapolating the pixels immediately to the left of the block.

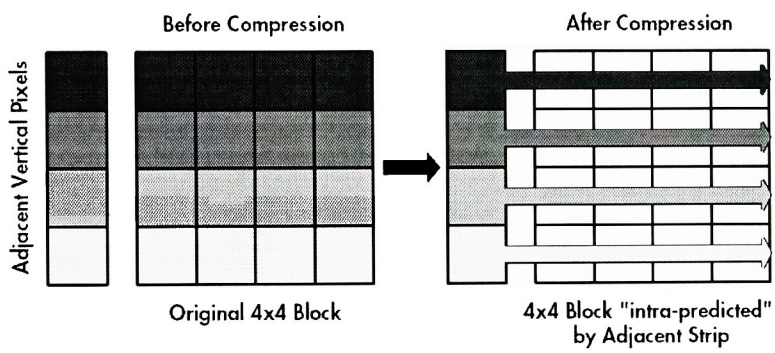

Figure 2.5: Intra estimation in H.264/AVC [1]

Transform

Instead of using a Discrete Cosine Transform (DCT), H.264/AVC uses an integer transform algorithm that approximates the DCT. Smaller block sizes of 4x4 pixels are encoded and decoded rather than 8x8 blocks, resulting in less blocking or ringing artifacts when compressed by the quantization stage and therefore resulting in a higher-quality image [1].

The integer transform used by H.264/AVC approximates the DCT, but uses substantially simpler arithmetic. MPEG-2 uses 32-bit floating point arithmetic for its transform unit, but the integer transform can be done using 16-bit integer arithmetic. The integer transform matrix coefficients have been adjusted to be integers or simple ratios (such as 1/2) so that no multiplications are needed in the transform stage (a scaling multiplication is done in the quantization stage). This means that all arithmetic for the transform can be

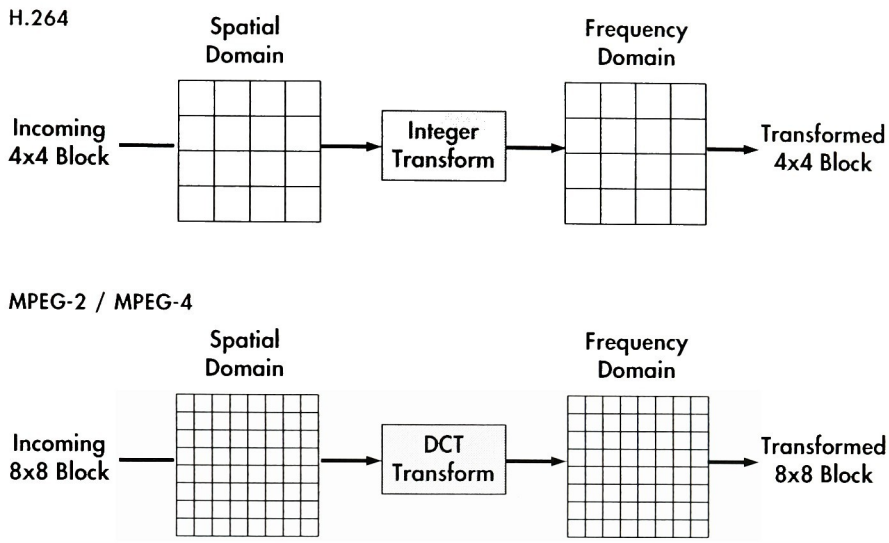

Figure 2.6: Transform stage changes from MPEG-2 [1]

accomplished using additions and shifts [16].

Quantization

The scalar quantizer used by H.264/AVC also avoids any division or floating-point arithmetic, enabling simple integer arithmetic to be used. The quantization stage incorporates the post- and pre-scaling factors for the integer transform. Because of the simpler algorithm, the quantization stage uses mostly shifts and additions and only one multiplication per coefficient, rather than the multiple multiplications per coefficient required by the MPEG-2 standard [1].

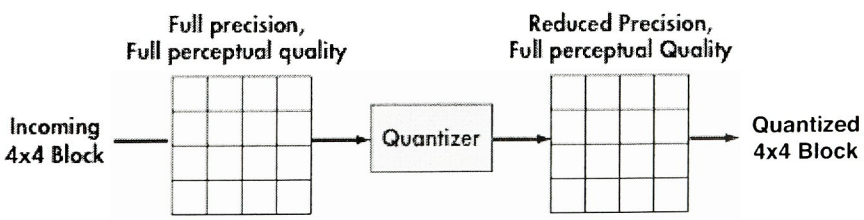

Figure 2.7: H.264/AVC quantization/rate control [1]

Characteristics	VLC	CAVLC	CABAC
Where it is used	MPEG-2	H.264/AVC Baseline Profile	H.264/AVC Main Profile
Probability distributions	One	Multiple	Adaptive
Leverages correlation between symbols	No	No	Yes
Non-integer code words	No	No	Yes

Table 2.2: H.264/AVC entropy coding comparison [1]

Deblocking Filter

An in-loop deblocking filter is implemented in the H.264/AVC standard to reduce artifacts produced by various compression techniques. The deblocking filter operates on both 16x16 pixel macroblocks and on 4x4 pixel block boundaries. For macroblocks, the filter reduces artifacts caused by different types of motion or intra estimation being used in adjacent blocks. The filter also helps to remove artifacts caused by transform/quantization of adjacent 4x4 blocks or from motion vector differences. The deblocking filter operates on the two pixels on either side of a boundary using a content adaptive non-linear filter [1].

Entropy Coding

H.264/AVC offers improved entropy coding to compress the final data bit stream. Instead of the older Variable Length Coding (VLC) used by MPEG-2, H.264/AVC offers two new entropy coding techniques, called Context Adaptive Binary Arithmetic Coding (CABAC) and Context Adaptive Variable Length Coding (CAVLC). CABAC uses arithmetic coding with non-integer codewords to allow greater bit rate reduction. It is capable of adapting to different probability distributions of data in order to better correlate the current bit patterns. CAVLC offers some of the entropy coding improvements of CABAC without all of the hardware complexity. CAVLC is a more adaptive version of VLC with multiple code tables that can be used based on the current context of the video data. A comparison of the entropy coding types is shown in Figure 2.2.

2.3.2 Performance Comparison

Because of the algorithm changes highlighted above, H.264/AVC achieves an image compression 30-70% better than that of MPEG-2. In a wide range of tests, H.264/AVC was shown to outperform MPEG-2 by an average of 63% in video streaming applications and by an average of 45% in entertainment-quality applications [20].

A sample performance comparison using the Foreman QCIF test video is shown in Figure 2.8. In this example, H.264/AVC can be seen to produce results with a significantly higher quality at a given bit-rate than any other type of video compression.

A plot of the bit-rate savings over a range of peak signal-to-noise ratio (PSNR) quality, given in Figure 2.9, shows that H.264/AVC produces a bit-rate savings of 55-65% over MPEG-2 for this test video sequence.

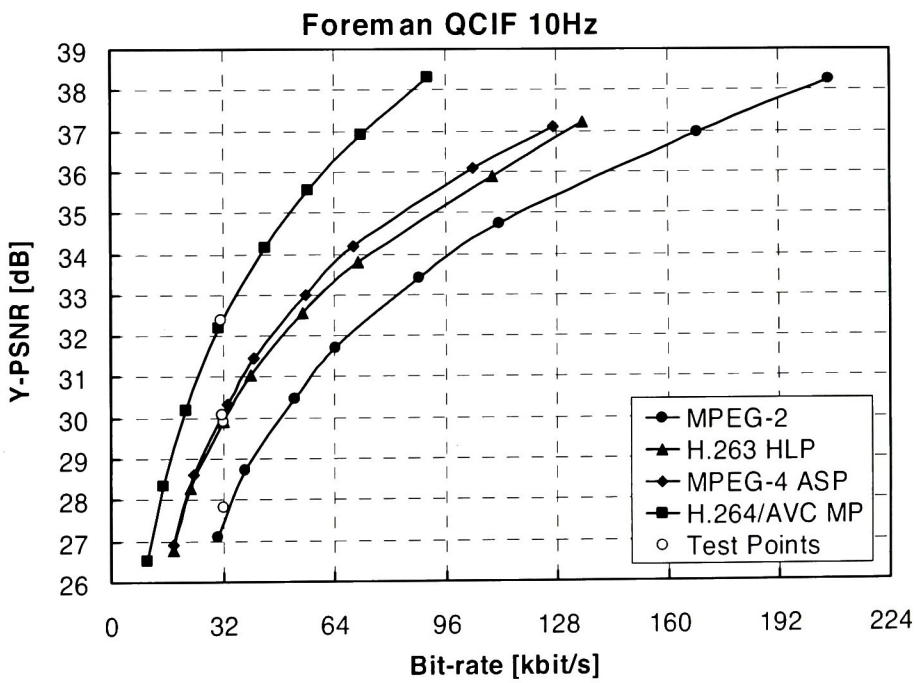

Figure 2.8: Video compression rate-distortion comparison using the Foreman QCIF video sequence [20]

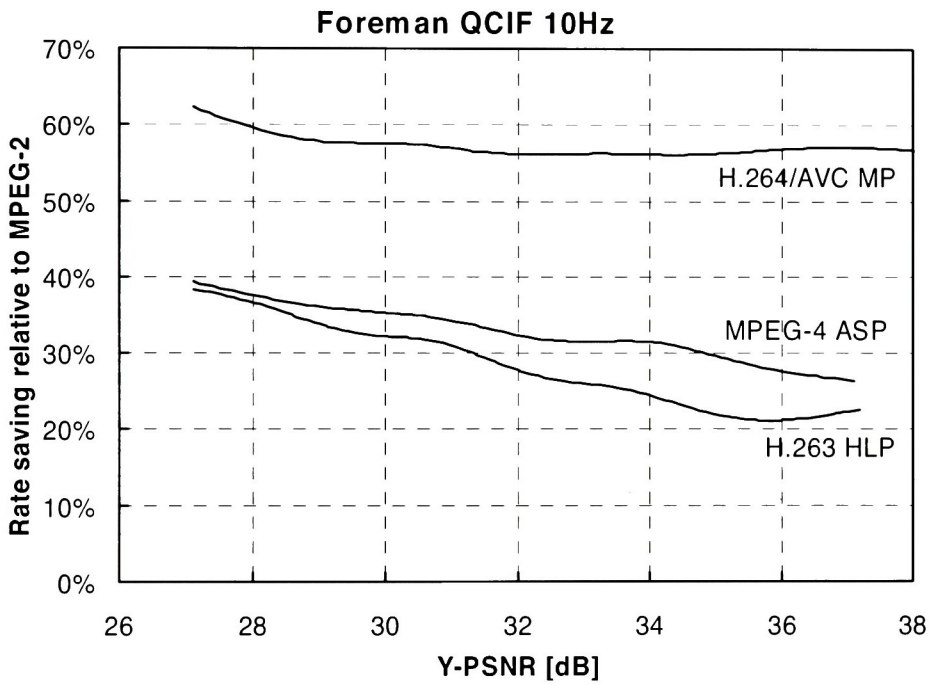

Figure 2.9: Video compression bit-rate saving comparison using the Foreman QCIF video sequence [20]

Chapter 3

Design – Algorithms Used

This chapter provides an overview of the algorithms used for each stage in the video decoder as defined for the Baseline Profile of the H.264/AVC standard ([8]).

3.1 Overall Video Decoder

The compressed video stream goes through the following functional blocks of the video decoder, also shown in the block diagram of Figure 3.1:

- Video Stream Parsing (Entropy Decoder and Run Length Decoder)
- Inverse Quantization
- Inverse Transform
- Intra Prediction
- Inter Prediction
- Deblocking Filter
- Frame Buffer

The video stream parsing stage uses entropy decoding to recover the video parameters. Run length decoding is then used to reconstruct the video residual coefficients as encoded

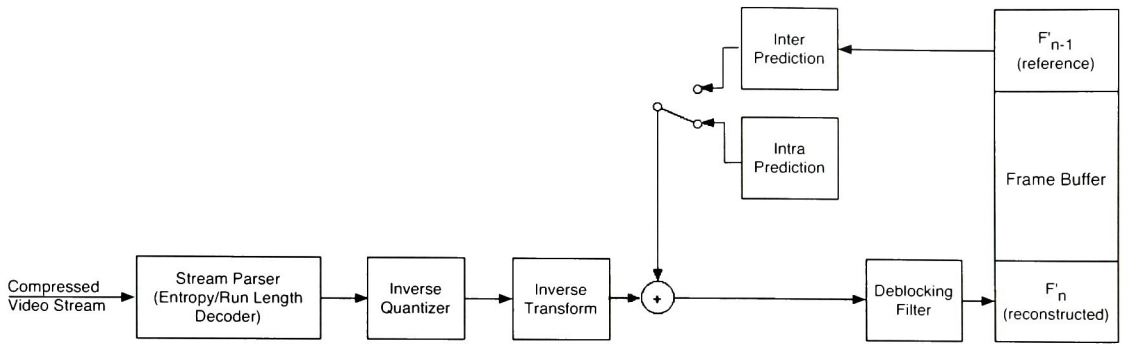

Figure 3.1: General dataflow of H.264/AVC decoder

in the frequency domain. This data then passes through the inverse quantization and inverse transform stages to recover the original residual coefficients in the space domain. Depending on the type of video slice (as determined in the header information), inter or intra prediction is used to estimate the content of the current video frame based on previous frames or macroblocks. The estimated data is summed with the decoded residual data to obtain the decoded video frame. The frame is then passed through a deblocking filter to smooth blocking artifacts and improve the picture quality. After a frame is decoded, it may be stored as a reference frame in the frame buffer for future use by the inter prediction process.

3.2 Video Stream Parsing

The video stream parsing stage parses video packets from the video stream, performs entropy decoding to recover the video header and data parameters, and reconstructs the video coefficients using run length decoding.

3.2.1 Network Abstraction Layer Parsing

All video data and header information is organized into network abstraction layer (NAL) units. The H.264/AVC network abstraction layer is designed to work well within a variety

Field	Description
forbidden_zero_bit	1-bit fixed (='0')
nal_ref_idc	2-bit unsigned number
nal_unit_type	5-bit unsigned number
rbsp_byte[]	Rest of bytes in NAL unit (the Raw Bit Sequence Payload (RBSP))

Table 3.1: NAL unit format

of different transport systems. Each NAL unit is a packet containing an integer number of bytes. Two NAL unit formats are defined: a byte stream format, which uses start code patterns to delineate NAL unit boundaries, and a packet-transport format, which relies on an outside packet transport protocol to differentiate between each NAL unit.

Each NAL unit is organized as shown in Table 3.1. The NAL unit type indicates the type of payload data that it contains. These NAL unit types, shown in Table 3.2, can be split into two types: video coding layer (VCL) and non-VCL NAL units. VCL units contain the actual video sample data. Non-VCL units contain associated header information that can apply to multiple video coding layer units. All the header information necessary for decoding the video data samples is contained in two types of NAL units: sequence parameter sets and picture parameter sets. Other supplemental enhancement information can also be sent using non-essential header unit types.

A coded video sequence will contain a single sequence parameter set, one or more picture parameter sets, and one or more access units containing a coded picture. Each access unit is structured as shown in Figure 3.2. The delimiters, the supplemental enhancement information (SEI), and the redundant coded picture are all optional. The only required part of the access unit is a primary coded picture, which consists of a set of VCL NAL units containing slices that together represent a video picture.

nal_unit_type	Content of NAL unit and RBSP syntax structure
0	Unspecified
1	Coded slice of a non-IDR picture
2	Coded slice data partition A
3	Coded slice data partition B
4	Coded slice data partition C
5	Coded slice of an IDR picture
6	Supplemental enhancement information (SEI)
7	Sequence parameter set
8	Picture parameter set
9	Access unit delimiter
10	End of sequence
11	End of stream
12	Filler data
13..23	Reserved
24..31	Unspecified

Table 3.2: NAL unit type codes [8]

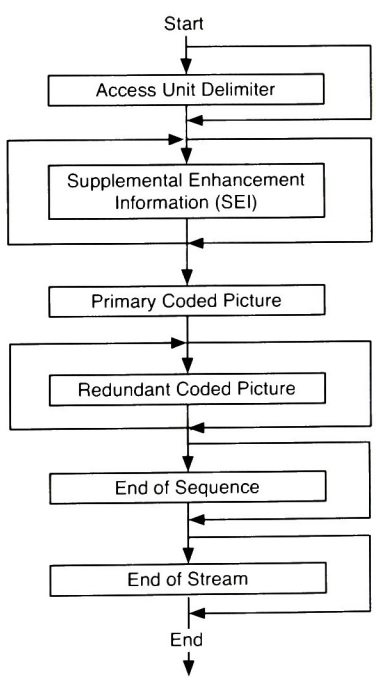

Figure 3.2: Access Unit structure [21]

Symbol	Element Type
ae(v)	Context-adaptive arithmetic entropy-coded syntax element
b(8)	Byte (8 bits)
ce(v)	Context-adaptive variable-length entropy-coded syntax element
f(n)	Fixed-pattern bit string using n bits
i(n)	Signed integer using n bits
me(v)	Mapped Exp-Golomb-coded syntax element
se(v)	Signed integer Exp-Golomb-coded syntax element
te(v)	Truncated Exp-Golomb-coded syntax element
u(n)	Unsigned integer using n bits
ue(v)	Unsigned integer Exp-Golomb-coded syntax element

Table 3.3: Syntax element descriptors [8]

3.2.2 Entropy Decoding

Each type of payload has a parsing function defined in the H.264/AVC standard. The information is received in an explicitly defined order with the entropy decoding method to use clearly defined. Each syntax element in the payload is encoded in one of the ways listed in Table 3.3.

The syntax elements fall into three basic categories: basic coding, Exp-Golomb coding, and context-adaptive coding. The basic elements are a simple byte, fixed pattern bit strings, and signed and unsigned integer types (*b(8)*, *f(n)*, *i(n)*, and *u(n)* respectively). Exp-Golomb elements have variable lengths and perform a table lookup for each type (*me(v)*, *se(v)*, *te(v)*, or *ue(v)*). The third type of syntax elements, context-adaptive codes, use one of two specially designed codes. The Baseline Profile implemented in this project uses a type of coding called Context Adaptive Variable Length Coding (CAVLC). Other profiles of the H.264/AVC standard support the Context Adaptive Binary Arithmetic Coding (CABAC) variable length coding standard for a higher coding efficiency.

In the Baseline Profile, video header information is encoded using basic and Exp-Golomb coding, while the video data parameters are encoded using CAVLC. In the Main Profile, both the header information and the video data can be encoded using CABAC.

After entropy decoding has been completed, the video data parameters are used by the run length decoder to reconstruct the video data coefficients.

3.2.3 Run Length Decoding

Run length coding is designed to encode zig-zag ordered residual video data in the most efficient manner. It takes advantage of several characteristics of quantized blocks of video data:

1. Each of the blocks of residual video contains mostly zeros.
2. The highest frequency (trailing) coefficients are often ± 1.
3. The number of non-zero coefficients in neighboring blocks is often correlated.
4. The magnitude of the coefficients tends to be larger at the beginning of the sequence of samples (the lower frequencies) and smaller toward the end of the sequence (the higher frequencies) [16].

Based on these characteristics, the run length code is able to compress the residual video coefficients so that the only coefficient levels that are explicitly transmitted in the video stream are non-zero coefficient levels (except for ± 1 levels at the end of the block). All other coefficients are reconstructed implicitly by knowing the total number of non-zero coefficients in the block, the total number of zeros occurring after the first non-zero coefficient, the signs of the trailing ± 1 coefficients, and the number of zeros preceding each non-zero coefficient. The parameters used to encode a block of video data are shown in Table 3.4.

Parsing the run length parameters is done in the following order by the entropy decoder. First, *coeff_token* is read and the *TotalCoeffs* and *TrailingOnes* variables are derived. Then the *trailing_ones_sign_flag* parameter is parsed *TrailingOnes* number of times to get the signs of the trailing one coeffients (in reverse order). A (*TotalCoeffs* - *TrailingOnes*) number of non-zero coefficients are then expected in the stream. The non-zero coefficient levels

Parameter	Description
coeff_token	Encodes the number of non-zero coefficients (TotalCoeff) and Trailing Ones (one per block)
trailing_ones_sign_flag	Sign of TrailingOne value (one per trailing one)
level_prefix	First part of code for non-zero coefficient (one per non-zero coefficient, excluding trailing ones)
level_suffix	Second part of code for non-zero coefficient (not always present)
total_zeros	Encodes the total number of zeros occurring after the first non-zero coefficient (in zig-zag order) (one per block)
run_before	Encodes number of zeros preceding each non-zero coefficient in reverse zig-zag order (one per run of zeros)

Table 3.4: CAVLC encoding parameters [15]

are reconstructed from the *level_prefix* and *level_suffix* parameters (*level_suffix* may not be present, depending on the current state of the decoding). The *total_zeros* parameter is then read from the stream, followed by as many *run_before* parameters as it takes to define the position and length of each run of zeros.

The block of coefficients is then reconstructed in reverse zig-zag order from the parameters decoded from the stream. The signed trailing ones are placed in the list, followed by the decoded non-zero levels. Each run of zeros is then inserted between the non-zero coefficients until all zeros have been inserted. The list of coefficients is then reversed and padded with zeros at the end to reconstruct the residual data. The coefficients are then placed in a 4x4 block in zig-zag order, as shown in Figure 3.3, to recover the original data matrix. This data matrix provides a more natural row-by-row ordering for processing by the transform unit.

3.3 Transform Unit

The transform unit performs the inverse quantization and inverse transform operations on macroblock data to recover the residual data.

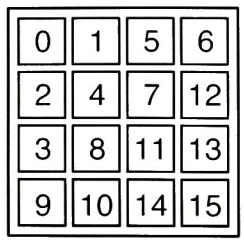

Figure 3.3: Zig-zag order of coefficients inside a block

3.3.1 Inverse Quantizer

Data entering the transform unit first passes through an inverse quantizer. This functional unit applies a scaling multiplication and some additional transformations to calculate the matrices of coefficients for the inverse transform stage.

Macroblock data is sent into the inverse quantizer in a specific scanning order of 4x4 data blocks, as shown in Figure 3.4. Each macroblock consists of 16 4x4 blocks of luma coefficients, 4 4x4 blocks of chroma Cb coefficients, and 4 4x4 blocks of chroma Cr coefficients. The first coefficient in each 4x4 block is referred to as a DC coefficient and all other coefficients in the block are referred to as AC coefficients. When the macroblock's coefficients are sent into the inverse quantizer, luma coefficients are sent first. If the macroblock has been encoded in 16x16 intra mode, the DC coefficients for the luma blocks are sent in as a separate 4x4 block prior to the AC coefficients. The chroma Cb and Cr coefficients are then sent into the inverse quantizer, with a 2x2 block of DC coefficients being sent before the AC coefficients for each chroma type.

The inverse quantization operation acts on DC and AC coefficients separately using the equations given below. At the end of the inverse quantization stage, the DC and AC coefficients are recombined to produce 4x4 blocks of coefficients for input to the inverse transform stage.

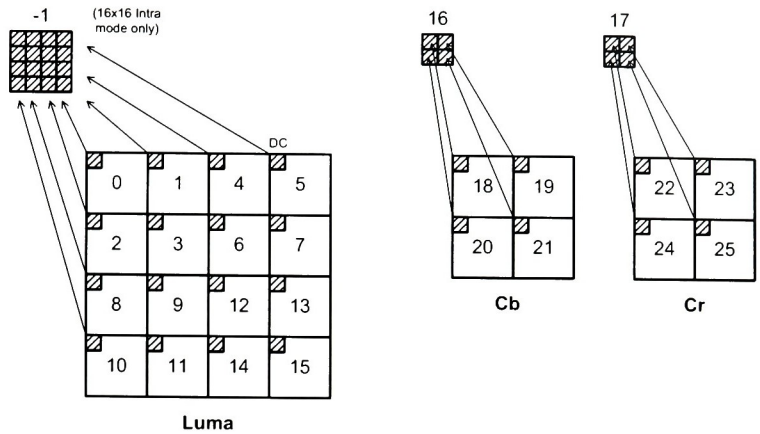

Figure 3.4: Macroblock data scanning order for inverse quantization [16]

AC coefficient transformation and rescaling

The inverse quantization stage performs the following base operation on all AC coefficients, where Q_{step} is based on the current value of the quantization parameter QP as defined in the video header information, $\boldsymbol{Z}$ is the incoming 4x4 matrix of coefficients, $\boldsymbol{Y}$ is the matrix of output coefficients, and i and j are the matrix indices:

$$Y_{ij} = Z_{ij} \cdot Q_{step} \tag{3.1}$$

The pre-scaling factors from the inverse transform stage (represented by PF) are also included in the inverse quantization stage, however, to simplify the inverse transform arithmetic. An additional pre-scaling factor of 64 is also included to decrease the error resulting from rounding. The inverse quantization stage therefore performs the following operation on incoming matrix $\boldsymbol{Z}$:

$$W_{ij} = Z_{ij} \cdot Q_{step} \cdot PF \cdot 64 \tag{3.2}$$

The scaling factors in the previous equation ($Qstep \cdot PF \cdot 64$) are constant for a given quantization level (given by QP) and a given coefficient matrix location. The equation can therefore be simplified to

QP	Positions (0,0),(2,0),(2,2),(0,2)	Positions (1,1),(1,3),(3,1),(3,3)	Other positions
0	10	16	13
1	11	18	14
2	13	20	16
3	14	23	18
4	16	25	20
5	18	29	23

Table 3.5: Value of the scaling matrix $\boldsymbol{V}$ for $0 \leq QP \leq 5$ [8]

$$W_{ij} = Z_{ij} \cdot V_{ij} \cdot 2^{floor(QP/6)} \tag{3.3}$$

where $\boldsymbol{V}$ is a matrix of constant scaling factors defined for each value of QP from 0 to 5, and $2^{floor(QP/6)}$ scales the matrix for $QP \geq 6$. The $\boldsymbol{V}$ matrix is defined by the values in Table 3.5.

DC coefficient transform and rescaling

The inverse quantization stage applies an additional transformation to DC coefficients prior to rescaling.

If the macroblock has been coded in Intra 16x16 mode, a 4x4 Hadamard transform is applied to the luma DC coefficients. The 4x4 Hadamard transform is given by:

$$\boldsymbol{W}_{QD} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} \boldsymbol{Z}_D \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) \tag{3.4}$$

where $\boldsymbol{Z}_D$ is the input block of luma DC coefficients and $\boldsymbol{W}_{QD}$ is the inverse quantization output prior to rescaling.

Rescaling of the luma DC coefficient block is then done using the following equations:

$$W'_{D(i,j)} = W_{QD(i,j)} \cdot V_{(0,0)} \cdot 2^{floor(QP/6)-2} (\text{if } QP \geq 12) \tag{3.5}$$

$$W'_{D(i,j)} = (W_{QD(i,j)} \cdot V_{(0,0)} + 2^{1-floor(QP/6)}) \gg (2 - floor(QP/6)) (\text{if } QP < 12) \tag{3.6}$$

where $V_{(0,0)}$ is a constant scaling factor defined for each value of QP from 0 to 5 in Table 3.5.

Chroma DC coefficients are recovered using a 2x2 Hadamard transform on the Cb and Cr 2x2 DC coefficient blocks:

$$\boldsymbol{W}_{QD} = (\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \boldsymbol{Z}_D \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}) \tag{3.7}$$

Rescaling of chroma DC coefficients is performed by:

$$W'_{D(i,j)} = W_{QD(i,j)} \cdot V_{(0,0)} \cdot 2^{floor(QP/6)-1} (\text{if } QP \geq 6) \tag{3.8}$$

$$W'_{D(i,j)} = (W_{QD(i,j)} \cdot V_{(0,0)}) \gg 1 (\text{if } QP < 6) \tag{3.9}$$

3.3.2 Inverse Integer Transform

This stage transforms the image data back from the frequency domain into the spatial domain. An inverse integer transform is performed to produce the final matrices of video residual data.

The integer transform used by H.264/AVC approximates the DCT, but adjusts the coefficients to be integers or simple ratios (such as $1/2$) that allow all arithmetic for the transform to be realized using only additions and shifts. The 4x4 inverse integer transform is given by the following equation:

$$\boldsymbol{X'} = \boldsymbol{C}_i^T(\boldsymbol{Y} \otimes \boldsymbol{E}_i)\boldsymbol{C}_i = \begin{bmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{bmatrix} \left(\begin{bmatrix} \boldsymbol{Y} \end{bmatrix} \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \right) \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix} \tag{3.10}$$

The scaling factors a and b are defined as:

$$a = \frac{1}{2} \tag{3.11}$$

$$b = \sqrt{\frac{2}{5}} \tag{3.12}$$

In an actual implementation of the inverse integer transform, the prescaling matrix factors are included in the inverse quantization stage. The integer transform stage takes the prescaled matrix $\boldsymbol{W}$ as an input, performs matrix multiplication on it using transform matrices $\boldsymbol{C}$ and $\boldsymbol{C}^T$, and divides the final values by 64 (by right shifting 6 places) to remove the scaling added by the inverse quantization stage. The implemented inverse transform stage thus performs the following operation:

$$\boldsymbol{X} = \begin{bmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} \boldsymbol{W} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix} \gg 6 \tag{3.13}$$

3.4 Intra Prediction

The intra prediction unit predicts the sample values of a block based on previously decoded blocks around it. This estimation is done in the spatial domain after the inverse transform has been performed. Several different intra modes are available:

- Intra_4x4 mode: Predicts each 4x4 luma block based on the bordering 4x4 blocks. This mode is used when coding parts of a picture with a significant level of detail.

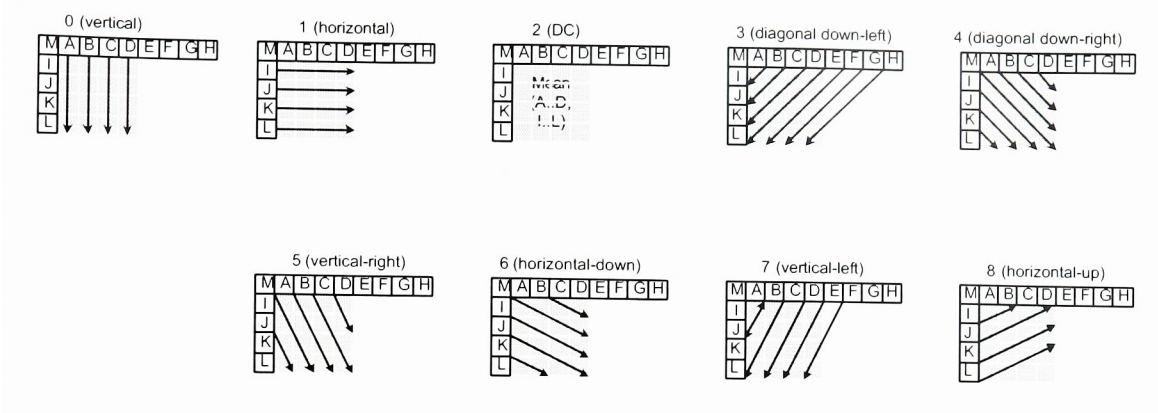

Figure 3.5: Intra 4x4 prediction modes [15]

- Intra_16x16 mode: Predicts each 16x16 luma macroblock based on the bordering 16x16 blocks. This mode is good for coding smooth areas of the picture.
- I_PCM mode: Bypasses the prediction and transform decoding stages and instead provides the original values of the samples. This can be useful for preserving picture quality in areas where the other prediction modes don't work well enough.

Chroma prediction is performed in a way similar to Intra_16x16 prediction.

Intra_4x4 Prediction

In Intra_4x4 mode, each 4x4 block is predicted based on samples from previously decoded 4x4 blocks above and to the left of the block. Nine different prediction modes are available, as described in Table 3.6 and shown visually in Figure3.5.

Intra_16x16 Prediction

The Intra_16x16 prediction mode operates on 16x16 macroblocks to provide a fast prediction that is useful in smooth regions of the picture. It has four modes: vertical, horizontal, and DC prediction, which are similar to their Intra_4x4 counterparts; and a plane prediction mode. The plane prediction mode works by fitting a linear plane function between the top-right and bottom-left corners of the macroblock, which can be used for smoothly varying

Mode	Description
Mode 0 (Vertical)	The upper samples A, B, C, D are extrapolated vertically
Mode 1 (Horizontal)	The left samples I, J, K, L are extrapolated horizontally
Mode 2 (DC)	All samples in the block are predicted by the mean of samples A...D and I...L.
Mode 3 (Diagonal Down-Left)	The samples are interpolated at a 45° angle between lower-left and upper-right.
Mode 4 (Diagonal Down-Right)	The samples are extrapolated at a 45° angle down and to the right.
Mode 5 (Vertical-Right)	Extrapolation at an angle of approximately 26.6° to the left of vertical (width/height = 1/2)).
Mode 6 (Horizontal-Down)	Extrapolation at an angle of approximately 26.6° below horizontal.
Mode 7 (Vertical-Left)	Extrapolation (or interpolation) at an angle of approximately 26.6° to the right of vertical.
Mode 8 (Horizontal-Up)	Interpolation at an angle of approximately 26.6° above horizontal.

Table 3.6: Intra_4x4 prediction modes [15]

areas of the picture.

In order to predict an entire 16x16 macroblock at one time, 32 samples are needed: the 16 samples in the macroblock directly above and the 16 samples in the macroblock directly to the left of the current macroblock. Prediction is done using the same type of algorithms as those described under Intra_4x4 prediction.

Chroma Prediction

Chroma prediction is performed on 8x8 chroma macroblocks. It has the same four prediction modes as Intra_16x16 prediction and operates identically on a smaller 8x8 block of data.

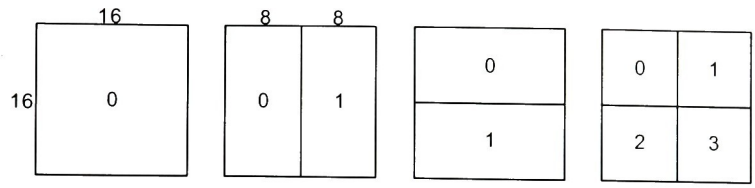

Figure 3.6: Macroblock partitions: 16x16, 8x16, 16x8, 8x8 [16]

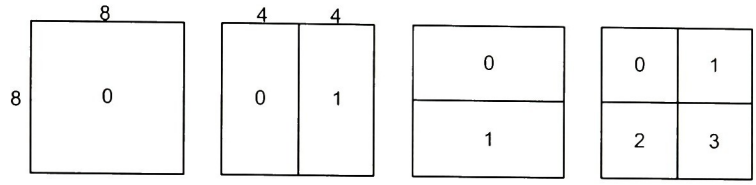

Figure 3.7: Sub-macroblock partitions: 8x8, 4x8, 8x4, 4x4 [16]

3.5 Inter Prediction

Inter prediction estimates a picture block based on data in a previously decoded frame and a motion vector that indicates how that portion of the picture moved. H.264/AVC supports a range of block sizes from 16x16 down to 4x4 and motion vectors that are calculated with a quarter-sample luma resolution. Since motion vectors are highly correlated between adjacent blocks, a method of predicting motion vectors is also used.

Tree structured motion compensation

Each macroblock can be split up into a wide range of differently sized blocks for use in motion compensation. The luminance component of the macroblock can be split up using four different partition methods consisting of one 16x16 partition, two 16x8 partitions, two 8x16 partitions, or four 8x8 partitions, as shown in Figure 3.6. If the macroblock is split into four 8x8 partitions, each 8x8 partition can be partitioned in one of four different ways: into one 8x8 sub-partition, two 4x8 sub-partitions, two 8x4 sub-partitions, or four 4x4 sub-partitions, as shown in Figure 3.7. This system of breaking up a macroblock into partitions of varying size is called tree structured motion compensation.

Each partition or sub-partition requires a separate motion vector that must be coded and

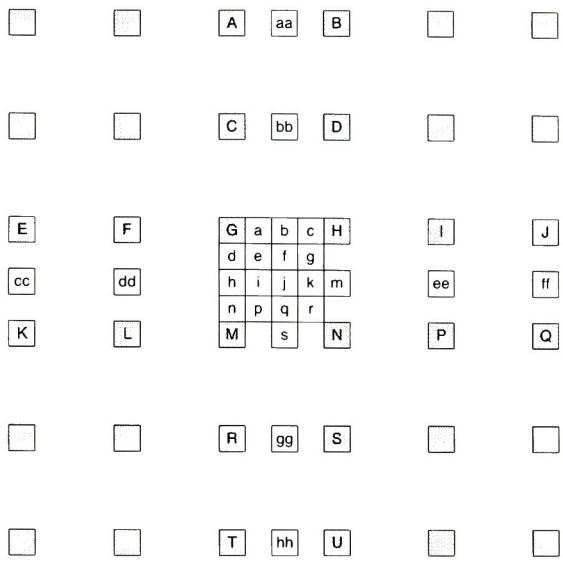

Figure 3.8: Interpolation of luma quarter-pixel positions [21]

transmitted in the data stream. Use of a large partition size means that fewer bits are required for signaling the choice of motion vector, but the predicted image is less accurate in areas of varying motion. Use of smaller partition sizes results in a more accurate prediction in areas of detailed motion, but requires more bits for transmitting the motion vector.

Each chroma component of the macroblock is partitioned in the same way as the luma component, except that the chroma partition sizes have half the horizontal and half the vertical resolution (e.g. a 8x16 partition in the luma component corresponds to a 4x8 partition in the chroma component).

Motion vectors

Each partition in a macroblock is predicted from an identically sized area in the reference frame. The offset between the two locations (the motion vector) is represented to the nearest quarter-pixel. Since the motion vector may point to a reference frame location that isn't an integer value, interpolation needs to be done on that area of the reference frame to derive quarter-pixel values.

Interpolation of half-pixel and quarter-pixel values is done using the pixel locations

indicated in Figure 3.8. The uppercase letters (A, B, C, etc.) indicate actual pixels in the reference frame. Lowercase letters indicate half-pixel or quarter-pixel locations.

Fractional sample interpolation is first done to derive the half-pixel samples. Interpolation is done by applying a 6-tap filter on full-pixel locations. The filtering is applied on a horizontal row of pixels to derive half-pixel samples, such as b, and then is applied on a vertical column of pixels to derive half-pixel samples such as h.

$$b_1 = (E - 5F + 20G + 20H - 5I + J) \tag{3.14}$$

$$h_1 = (A - 5C + 20G + 20M - 5R + T) \tag{3.15}$$

After the intermediate values b_1 and h_1 are calculated, they are then rounded to produce the final half-pixel samples b and h:

$$b = (b_1 + 16) \gg 5 \tag{3.16}$$

$$h = (h_1 + 16) \gg 5 \tag{3.17}$$

The sample at half-pixel position j is obtained by interpolating other half-pixel samples:

$$j = ((cc - 5dd + 20h_1 + 20m_1 - 5ee + ff) + 512) \gg 10 \tag{3.18}$$

The samples at quarter-pixel positions a, c, d, n, f, i, k, and q are then calculated by averaging the two nearest samples at integer and half-pixel positions, such as:

$$a = (G + b + 1) \gg 1 \tag{3.19}$$

The samples at quarter-pixel positions e, g, p, and r are calculated by averaging the two nearest samples at half-pixel positions in the diagonal direction, such as:

$$e = (b + h + 1) \gg 1 \tag{3.20}$$

Chroma motion vectors make use of eighth-sample intervals between integer pixel locations. Interpolation of a sub-sample position a is done using a linear combination of the four full-pixel sample positions A, B, C, and D immediately surrounding the sample to be calculated (these positions correspond to luma locations G, H, M, and N respectively):

$$a = (((8-d_x)\cdot(8-d_y)A + d_x\cdot(8-d_y)B + (8-d_x)\cdot d_y C + d_x\cdot d_y D) + 32) \gg 6 \quad (3.21)$$

Motion vector prediction

Since it can take a large number of bits to transmit a complete motion vector for each partition, motion vector prediction has been implemented to predict a partition's motion vector based on the motion vectors of surrounding partitions. This allows only the differences between the predicted motion vector and the actual motion vector to be sent as part of the data stream. .

The predicted motion vector *MVp* is derived by examining the partitions or sub-partitions immediately surrounding the current partition (called E). The following rules are used (where the topmost partition to the left of E is called A, the leftmost partition immediately above E is called B, and the partition immediately to the right and above E is called C):

1. For all partitions of E excluding 16x8 and 8x16 sizes, *MVp* is the median of the motion vectors for A, B, and C.

2. For 16x8 partitions, *MVp* for an upper 16x8 partition is predicted from B and *MVp* for a lower 16x8 partition is predicted from A.

3. For 8x16 partitions, *MVp* for a left 8x16 partition is predicted from A and *MVp* for a right 8x16 partition is predicted from B.

Skipped macroblocks are treated as 16x16 partitions. If a partition used for prediction is unavailable, a predicted motion vector of all zeros is produced.

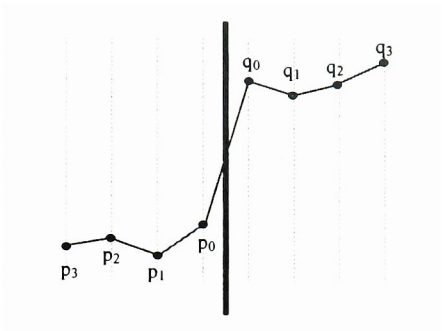

Figure 3.9: 1-dimensional visualization of block edge where filtering would be turned on [11]

3.6 Deblocking Filter

The deblocking filter performs in-loop filtering to reduce blocking artifacts created by image partitioning and quantization. After each macroblock is decoded, the deblocking filter compares the edge chrominance and luminance values of each 4x4 block with its adjacent blocks. Depending on the size of the differences between adjacent coefficients, a different level of filtering can be chosen, from no filtering to strong filtering. The encoded video stream can also include special information to increase or decrease the amount of deblocking filtering to be applied. The deblocking filter is adaptive on the individual sample level, the block-edge level, and slice level. The amount and type of filtering is chosen based on filtering coefficients determined by analysis of each of these different levels [11].

Sample-Level Adaptivity

On the individual sample level, analysis occurs to distinguish between true image edges and blocking artifacts. A line of samples in two adjoining 4x4 blocks can be designated as $p_3, p_2, p_1, p_0, q_0, q_1, q_2, q_3$, with the block boundary occurring between p_0 and q_0 as shown in Figure 3.9. Up to three luminance sample values and one chrominance value on each side of the boundary is permitted to be modified by the filtering.

Two threshold values, α and β, are derived from a table lookup containing empirically determined visually pleasing values. The index for the table lookup is based on the average

quantization parameter (*QP*), as well as encoder-selected offset values Offset_A and Offset_B. The table index values are calculated as:

$$\text{Index}_A = \text{Min}(\text{Max}(0, \text{QP} + \text{Offset}_A), 51) \tag{3.22}$$

$$\text{Index}_B = \text{Min}(\text{Max}(0, \text{QP} + \text{Offset}_B), 51) \tag{3.23}$$

The decision to filter a line of samples is based on whether all three of the following conditions are true:

$$|p_0 - q_0| < \alpha(\text{Index}_A) \tag{3.24}$$

$$|p_1 - p_0| < \beta(\text{Index}_B) \tag{3.25}$$

$$|q_1 - q_0| < \beta(\text{Index}_B) \tag{3.26}$$

In these equations, α and β are threshold variables calculated by a table lookup using Index_A and Index_B. Since α is generally much larger than β, only samples that have a large change on the boundary without any large changes on either side will be filtered. Since α and β depend on QP, more filtering will be performed as the coding error increases with increased quantization.

Edge-Level Adaptivity

All luminance block edges in an image are analyzed to determine a Boundary-Strength (*Bs*) filtering parameter. Table 3.7 shows the conditions used (starting at the top and proceeding downward) to determine the *Bs* value for a given block edge. When filtering is performed, *Bs* determines the strength of filtering used. A *Bs* value of 0 results in no filtering being performed; a *Bs* value of 4 results in a special mode for the strongest filtering; and *Bs* values between 1 and 3 are used as part of a standard filtering algorithm.

The *Bs* values reflect the fact that the strongest blocking artifacts occur due to intra and prediction error coding, while block motion compensation has a smaller effect in producing

Block modes and conditions	***Bs***
One of the blocks is Intra coded and the edge is a macroblock edge	4
One of the blocks is Intra coded	3
One of the blocks has coded residuals	2
Difference of block motion is $\geq$ 1 luma sample distance	1
Motion compensation from different reference frames	1
Else	0

Table 3.7: Filter strength parameter for each coding mode [11]

artifacts. *Bs* values for chrominance block edges are not calculated, but rather use the values calculated for corresponding luminance block edges.

Slice-Level Adaptivity

Additional control over the deblocking filter can be achieved using slice-level offsets. Values of Offset_A and Offset_B, used to adjust the values of α and β, can be selected by the encoder and stored with the slice header. This added control allows the encoder to optimize the quality of the video for the particular video application used. For high quality images, small spatial details can be better preserved by including negative offsets to cause less filtering. Filtering can be increased for low resolution content by choosing positive offsets on the slice level.

Order of Filtering

A pre-defined order of filtering must be followed in order to avoid mismatch between the video encoder and decoder. Since filtering is done in place, with filtered values being used as inputs to the next filtering operation, following a consistent order is essential to preserve image quality.

Filtering is done on each 16x16 macroblock, with macroblocks being filtered in raster-scan order. Each macroblock first has its vertical edges filtered (in left to right order), then

has its horizontal edges filtered (in top to bottom order). Luminance macroblocks first have their left edges filtered, followed by the three internal vertical edges. The top edge is then filtered, followed by the three internal horizontal edges. A similar order is followed with chrominance macroblocks, except the smaller 8x8 chrominance macroblock only requires filtering on one external edge and one internal edge in each direction.

Two different modes of filtering are defined, depending on whether *Bs* is between 1 and 3, or whether *Bs* has a value of 4. For both modes, two additional conditions are used to determine the extent of the filtering:

$$|p_2 - p_0| < \beta(\text{Index}_B) \tag{3.27}$$

$$|q_2 - q_0| < \beta(\text{Index}_B) \tag{3.28}$$

When both conditions are true, stronger filtering can be used because there is little intensity variation on either side of the edge boundary.

Filtering when *Bs* = 1 to 3

Luminance filtering, which can modify up to three values on each side of the boundary, calculates filtered values using

$$p'_0 = p_0 + \Delta_0 \text{ and} \tag{3.29}$$

$$q'_0 = q_0 + \Delta_0 \tag{3.30}$$

where Δ_0 is derived by calculating an initial Δ_{0i} value and clipping it to limit the strength of the low-pass filtering:

$$\Delta_{0i} = (4(q_0 - p_0) + (p_1 - q_1) + 4) \gg 3 \tag{3.31}$$

Filtering of edge values p_0 and q_0 has filter parameter Δ_0 clipped to be

$$\Delta_0 = \text{Min}(\text{Max}(-c_0, \Delta 0i), c_0) \tag{3.32}$$

where c_0 is set equal to a table lookup value, then incremented by 1 for each of the edge intensity equations that hold true. This results in stronger filtering being applied to edge samples with small intensity changes on either side of the boundary.

Internal samples p_1 and q_1 are only modified if their corresponding intensity equation holds true. If surrounding intensity changes are small, then the filtered values are calculated as

$$p_1' = p_1 + \Delta_{p1} \text{ and} \tag{3.33}$$

$$q_1' = q_1 + \Delta_{q1}, \tag{3.34}$$

where Δ_{p1} and Δ_{q1} are derived by calculating initial Δ_{p1i} and Δ_{q1i} values and clipping them:

$$\Delta_{p1i} = (p_2 + ((p_0 + q_0 + 1) \gg 1) - 2p_1) \gg 1 \tag{3.35}$$

The value of Δ_{q1i} is obtained by substituting q_2 and q_1 for p_2 and p_1 in the above equation. The final clipping parameters are calculated as

$$\Delta_{p1} = \text{Min}(\text{Max}(-c_1, \Delta p1i), c_1) \text{ and} \tag{3.36}$$

$$\Delta_{q1} = \text{Min}(\text{Max}(-c_1, \Delta q1i), c_1), \tag{3.37}$$

where c_1 is obtained by a table lookup using Index_A and the current *Bs* value.

Chrominance filtering is done in a similar way, except that only p_0 and q_0 values may be modified. The clipping parameter c_0 is set equal to $c_1 + 1$, removing the need to evaluation the intensity conditions or access sample values p_2 and q_2.

Filtering when $Bs = 4$

A special case of very strong filtering is used for boundaries between macroblocks that have been intra coded. This reduces blocking artifacts caused when intra coding nearly uniform image areas.

The strong filter is used for luminance filtering when the condition is satisfied that

$$|p_0 - q_0| < (a \gg 2) + 2 \tag{3.38}$$

and the previously used intensity equations hold true for surrounding values. When all conditions are true, filtered values are calculated as:

$$p'_0 = (p_2 + 2p_1 + 2p_0 + 2q_0 + q_1 + 4) \gg 3 \tag{3.39}$$

$$p'_1 = (p_2 + p_1 + p_0 + q_0 + 2) \gg 2 \tag{3.40}$$

$$p'_2 = (2p_3 + 3p_2 + p_1 + p_0 + q_0 + 4) \gg 3 \tag{3.41}$$

The q values are calculated by substituting q for p in the above equations. For chrominance filtering, if either the intensity equation or the strong filter condition is false, only p_0 is modified, leaving p_1 and p_2 unchanged:

$$p_0 = (2p_1 + p_0 + q_1 + 2) \gg 2 \tag{3.42}$$

Again, q values are obtained by substituting q for p in the above equation.

3.7 Frame Buffer

The final video sequence after filtering is stored into the frame storage interface. From there it can be displayed to the video output or stored as a reference frame for future use in motion compensation based picture reconstruction.

Operation	Reference Picture List				
Initial state	-	-	-	-	-
Decode frame 250	250	-	-	-	-
Decode 251	251	250	-	-	-
Decode 252	252	251	250	-	-
Decode 253	253	252	251	250	-
Assign 251 to LongTermPicNum 0	253	252	250	0	-
Decode 254	254	253	252	250	0
Assign 253 to LongTermPicNum 4	254	252	250	0	4
Decode 255	255	254	252	0	4

Table 3.8: Example of reference picture list updates [15]

Reference picture management

After each frame is decoded, it is marked as used for short-term reference, used for long-term reference, or unused for reference. Only frames transmitted with a nonzero *nal_ref_idc* can be used for reference. Each slice header contains a *long_term_reference_flag* that determines whether the frame should be marked as used for long-term reference or used for short-term reference. Once marked, a frame continues to be marked as used for reference until the next instantanious decoding refresh (IDR) frame is decoded. Whenever an IDR frame is decoded, all previous frames are marked as unused for reference.

Before each new slice is decoded, a reference picture list is built of all the frames currently available for reference. The reference picture list is built as a sliding window, where the frame with the highest assigned *PicNum* (the most recently decoded frame) is placed at the beginning of the list and pushes the older frames toward the end of the list. Long-term reference frames are assigned a *LongTermPicNum* and stored in ascending order after the short-term frames. The reference picture list has a specific window size; if it becomes full, the short-term reference frame with the lowest *PicNum* will be pushed off to accomodate a new frame. Long-term reference frames remain permanently on the list until being marked as unused for reference. The encoder may send special adaptive memory control commands to change the order or the marking of reference frames.

An example of reference picture management is shown in Table 3.8. In this example, frames with *PicNum* values of 250, 251, 252, and 253 are decoded and placed on the reference picture list in reverse order. The video header then contains a command to assign *LongTermPicNum* 0 to frame 251, causing the frame to be marked as "used for long-term reference" and moved to the end of the reference list. Similarly, frame 254 is decoded and 253 is assigned *LongTermPicNum* 4. At this point, the reference picture list is full, since it currently has a window size of five. When frame 255 is decoded and placed on the reference picture list, the oldest short-term picture, 250, is taken off the list to make room for the new frame.

The algorithms described in this chapter were used as the basis for implementing the H.264/AVC decoder. Details of how the algorithms were implemented are given in the next chapter.

Chapter 4

Implementation – Behavioral and Synthesizable VHDL

This chapter describes the behavioral VHDL model implementation of the H.264/AVC video decoder. It also dicusses the synthesizable hardware design of the transform unit and the deblocking filter stages.

4.1 Overall Video Decoder

A detailed diagram of the decoder dataflow is shown in Figure 4.1. As the video sequence is decoded, header information must be stored to provide control of each of the decoding units. Intermediate data must also be stored after each decoding unit to provide the input to the next component. The transform unit (inverse quantizer and inverse transform), inter and intra prediction stages, and deblocking filter all operate on one macroblock of data at a time. Additional data and header information from previously decoded macroblocks also need to be stored for use by the intra prediction unit and deblocking filter. Similarly, the inter prediction unit needs data and header information from previously decoded and stored reference frames.

The complete decoder was modeled using behavioral VHDL. The VHDL code was written to implement the algorithms given by the H.264/AVC standard in a straightforward way that could be easily used to help understand the details of the standard. The code

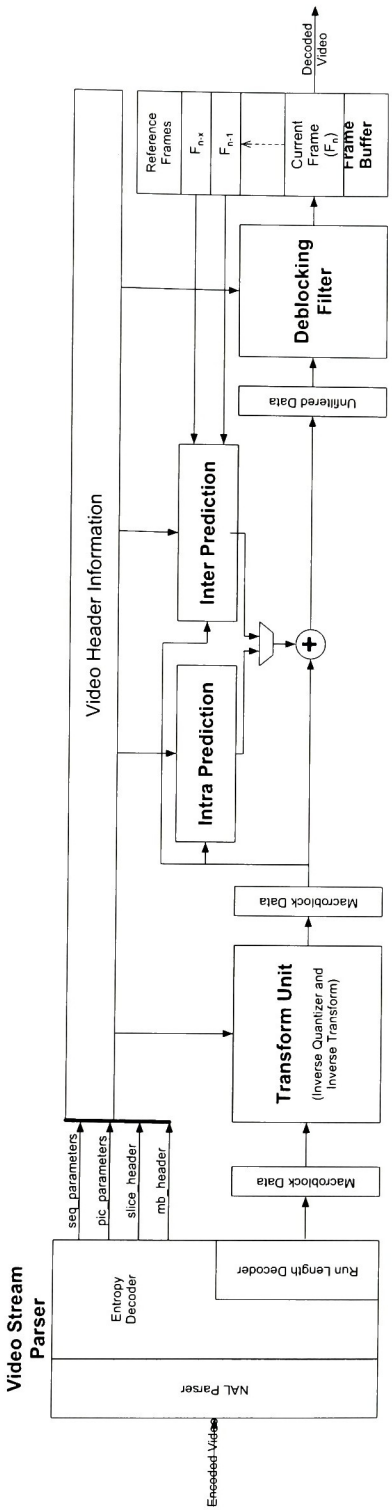

Figure 4.1: Detailed dataflow of the H.264/AVC decoder

was broken into procedures that match the sections and subsections of the standard, and commenting was used to refer to the specific equations, tables, and clauses that were being implemented.

A synthesizable design for the transform and deblocking filter units was then implemented based on the algorithms described in the behavioral code. Hardware designs from various papers were also examined to evaluate possible architectures to use in implementation.

4.2 Video Stream Parsing

This first stage of the decoder extracts the data from the incoming stream and decodes it into the video header information and data coefficients used by subsequent stages. The algorithm used is described in more detail in Chapter 3.

4.2.1 Network Abstraction Layer Parsing

Behavioral Model

The top-level behavioral model of NAL unit parsing is implemented in the *stream_parser.vhdl* file. This top-level consists of two procedures:

readNAL() Takes an input file and parses it into an array of NAL units. Each NAL unit consists of a header containing the *nal_ref_idc* and the *nal_unit_type* and a payload consisting of an array of the bits contained in the NAL unit.

parseNAL() Loops through each NAL unit and uses the *nal_unit_type* to call the appropriate parsing procedure on the payload. Returns the *seq_parameter_set*, an array of *pic_parameter_sets*, and an array of slices decoded from the NAL unit payloads. Currently supported types and their parsing procedures are shown in Table 4.1.

nal_unit_type	**Content of the NAL unit and procedure to call**
1	Coded slice of a non-IDR picture (e.g. a P-slice) *slice_layer_without_partitioning_rbsp()*
5	Coded slice of an IDR picture (e.g. an I-slice) *slice_layer_without_partitioning_rbsp()*
7	Sequence parameter set *seq_parameter_set_rbsp()*
8	Picture parameter set *pic_parameter_set_rbsp()*

Table 4.1: Currently supported NAL unit types and associated parsing procedures

4.2.2 Entropy Decoding

Behavioral Model

Each of the parsing procedures called by the NAL parser is contained in the *parameter_utils.vhdl* file. Each procedure takes as an input the NAL payload data array and the current position in the array. This allows the procedures to simulate the operation of parsing a data bitstream as it arrives; while the complete set of data has already been copied from the video file, each procedure is only allowed to see the next data available in the "stream".

Each procedure uses the header information that has already been decoded from the stream in order to make choices about what data to expect next in the stream. Each data field that is parsed from the stream has a specific type of decoding that needs to be applied, but often that decoding involves reading the stream until some sequence of bits is encountered. In order to accomodate this, each type of decoding was implemented in a procedure that took in the data array and the current position, incremented the position until it extracted the data it needed from the stream, and returned the new position along with the decoded field. The different field decoding procedures are shown in Table 4.2.

The top-level slice layer parsing procedure shown in Table 4.1 calls a number of sub-procedures to handle parsing the different parts of a slice. A hierarchy of the procedures

Procedure	Field Element Type
parse_b()	Byte (8 bits)
parse_f()	Fixed-pattern bit string using n bits
parse_i()	Signed integer using n bits
parse_u()	Unsigned integer using n bits
parse_me()	Mapped Exp-Golomb-coded syntax element
parse_se()	Signed integer Exp-Golomb-coded syntax element
parse_te()	Truncated Exp-Golomb-coded syntax element
parse_ue()	Unsigned integer Exp-Golomb-coded syntax element
parse_ce()	Context-adaptive variable-length entropy-coded syntax element

Table 4.2: Field parsing procedures

called is shown here:

slice_layer_without_partitioning_rbsp()

slice_header()

ref_pic_list_reordering()

pred_weight_table()

dec_ref_pic_marking()

slice_data()

macroblock_layer()

mb_pred()

sub_mb_pred()

residual()

residual_block_cavlc()

4.2.3 Run Length Decoding

Behavioral Code

This unit reconstructs the residual video coefficients for each macroblock. The coefficients may be Intra 16x16 DC, Intra 16x16 AC, Luma, Chroma DC, or Chroma AC coefficients

encoded using the CAVLC variable length coding standard.

The behavioral model that performs CAVLC decoding and run length decoding is contained within the *parameter_utils.vhdl* file. Two main procedures make up the model:

residual() Loops through each 4x4 block in a macroblock, determines what type of residual data has been encoded (if any), and calls *residual_cavlc()* to decode the run length parameters and use run length decoding to obtain the data values for the 4x4 block of residual data.

residual_cavlc() Parses the CAVLC parameters from the data stream and uses them to reconstruct a complete block of residual data using run length decoding. Uses the algorithm described in Chapter 3.

The CAVLC parameters and zig-zag ordered video data are parsed into the macroblock's *LumaValues* and *ChromaValues* data structures by *residual_cavlc()*. The data is then reordered by *residual()* into a 4x4 block and placed into the appropriate macroblock structure (depending on the type of data that was expected): *Intra16x16DCCoeff*, *Intra16x16ACCoeff*, *LumaCoeff*, *ChromaDCCoeff*, or *ChromaACCoeff*.

4.3 Transform Unit

The transform unit performs inverse quantization and inverse transform operations to recover the original video residual data. It takes the frequency-domain coded residual coefficients output from the VLC decoder and recovers the original space-domain residual coefficients.

4.3.1 Overall Transform Unit

An identical transform unit interface was used for both the behavioral and synthesizable models. The models were designed to use the same I/O interface in order to make it easy to

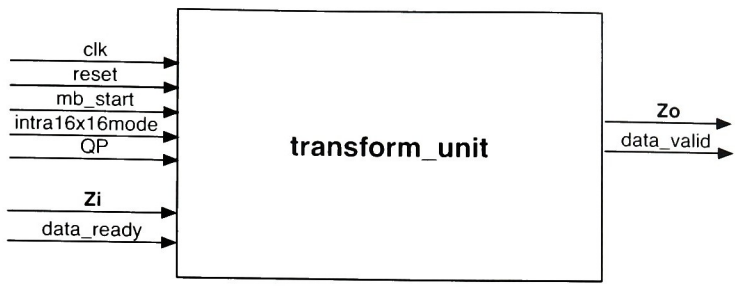

Figure 4.2: Transform unit interface for synthesizable design

interchange them in simulation. The detailed transform unit interface is shown in the block diagram of Figure 4.2.

Behavioral Model Design

The behavioral model of the transform unit was implemented as a top-level behavioral architecture in the *transform_unit.vhdl* file. The following four processes modeled all operations needed for the transform unit:

readInputCoefficients This process waited for the *mb_start* input to go high, then proceeded to collect a macroblock's worth of input coefficients and store them. Four data coefficients were read from the *Zi* input every clock edge when *data_ready* was high. The macroblock scanning process described in Figure 3.4 was followed to collect the Intra 16x16 DC and AC coefficients (if *intra16x16mode* was high), the Luma coefficients (if *intra16x16mode* was low), the Chroma Cb DC and AC coefficients, and the Chroma Cr DC and AC coefficients. All coefficients were stored in VHDL signal types that represented 4x4 matrices of data.

performInverseQuantization This process waited until all input coefficients for the macroblock had been read, then performed inverse quantization on the coefficients. The inverse quantization equations described in Chapter 3 were performed using custom VHDL functions that implemented mathematical operations on 4x4 matrices.

performInverseTransform This process took the output of the inverse quantization process and applied the inverse transform as shown in the equations of Chapter 3. The mathematical operations were performed using custom VHDL functions that implemented 4x4 matrix arithmetic.

writeOutputCoefficients After the inverse quantization and inverse transform had been applied, the coefficients in the macroblock were output by this process. All AC and DC coefficients were combined so that the output consisted of 16 4x4 Luma blocks, 4 Chroma Cb and 4 Chroma Cr 4x4 blocks. On each clock edge until the macroblock was finished, a column of four coefficients was output on *Zo* and the *data_valid* output was set high.

Synthesizable Digital Design

The design of the transform unit datapath is shown in the block diagram of Figure 4.3. The inverse quantization operation was split between two modules, one for AC coefficients and one for DC coefficients. Because the AC and DC coefficients are transmitted in different blocks, a register had to be inserted between the inverse quantization and inverse transform units to collect the data and group the final 4x4 coefficient blocks. As soon as a complete 4-coefficient row of coefficients is reconstructed in the macroblock register, the row is sent to the inverse transform unit for transformation. The inverse transform unit then outputs post-transform coefficients one column at a time.

4.3.2 Inverse Quantization

The inverse quantization stage prepares the coefficients for the inverse transform stage.

Synthesizable Digital Design

The inverse quantization operation was split into two modules: one that handles AC data coefficients, and one that handles DC coefficients. This was done because of the differences

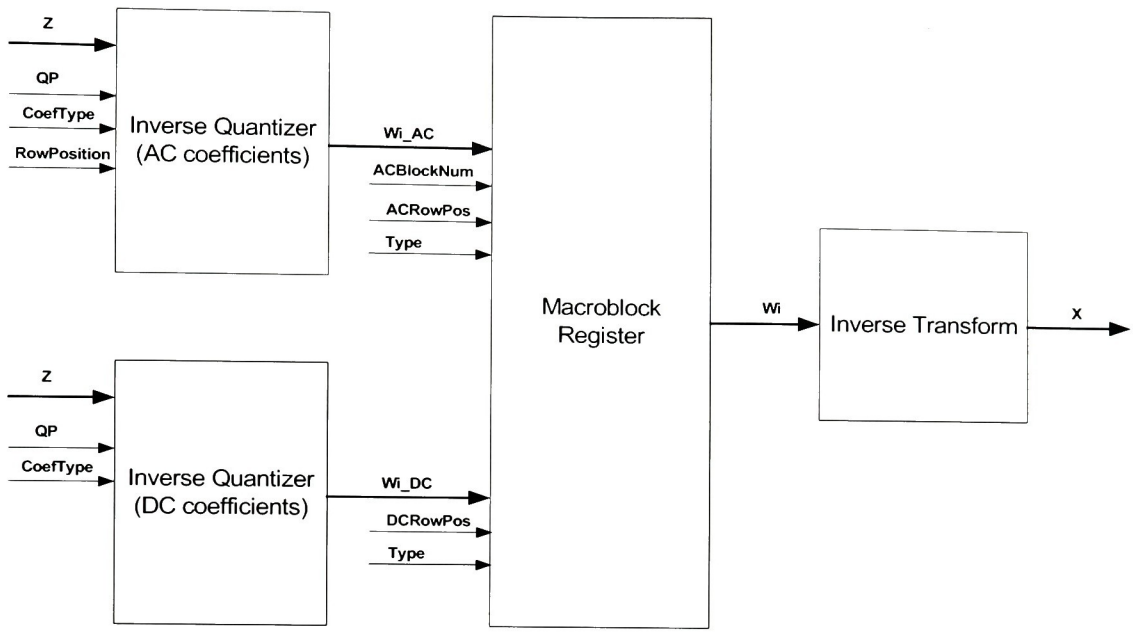

Figure 4.3: Transform unit datapath for synthesizable design

in how the different coefficient types are handled. AC coefficients are scalar multiplied by a value found in a table lookup (based on the current QP level). DC coefficients must go through a matrix multiplication before a further scaling operation. Because of this, the AC coefficient operations can be performed in a single cycle, while DC coefficients must pass through a five-cycle pipeline.

The AC inverse quantization unit is shown in Figure 4.4. Input *Zi* receives a 4 x 16 bit row of coefficients. A table lookup is performed based on the current block row number and the *QP* value to find the scaling factors for each coefficient. Each of the four input coefficients is then effectively multiplied individually by their respective scaling factor and the resulting coefficients are output as *Wi*.

The overall inverse AC quantization operation is performed in a single clock cycle. The multiplication is actually performed as a series of shifts and additions, simplifying the logic required. The input *Zi* coefficients are shifted to produce different factors of two for each coefficient; the desired set of factors are then selected (based on the row number and *QP* value) to be added together to effectively perform a table lookup and multiplication. The

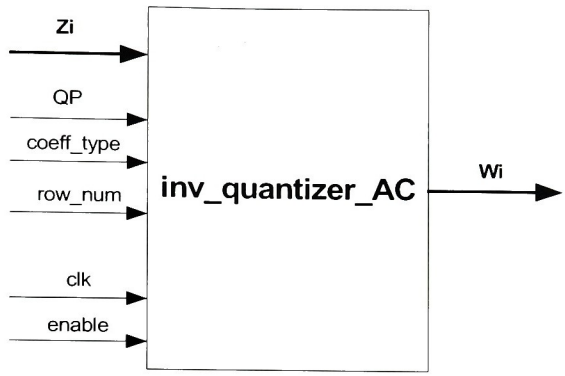

Figure 4.4: Inverse Quantizer AC interface for synthesizable design

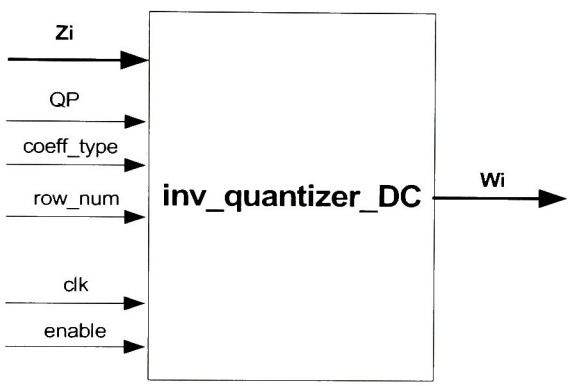

Figure 4.5: Inverse Quantizer DC interface for synthesizable design

delay through the unit is approximately the delay of two 16-bit adders.

The DC inverse quantization unit is shown in Figure 4.5. Input *Zi* receives a 4 x 16 bit row of coefficients. The DC unit can handle both 4x4 luma DC blocks and 2x2 chroma DC blocks by changing the value of the *coeff_type* input. To enter a 2x2 block of chroma coefficients, *coeff_type* must be set to 1 and the coefficients must be input as if they were the upper left 2x2 coefficients of a 4x4 block (setting all other coefficients to zero).

Each block of DC coefficients is passed through a Hadamard transform and then scaled before being output as 4x16-bit column *Wi*. In order to simplify the datapath control, both luma and chroma coefficients will be output five cycles after being input.

The Hadamard transform is implemented using adders and twos complement units, according to the butterfly diagram of Figure 4.6.

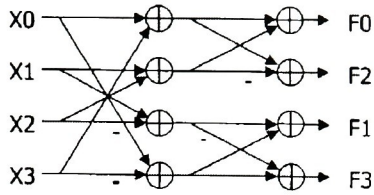

Figure 4.6: Hadamard Transform butterfly diagram [18]

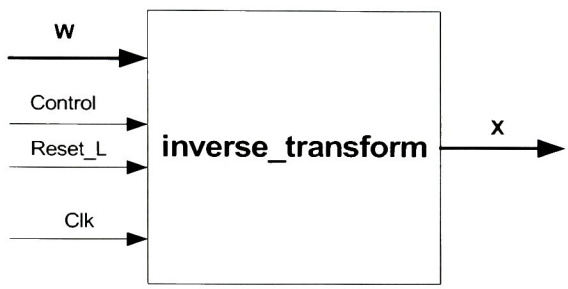

Figure 4.7: Inverse Transform interface for synthesizable design

The complete transform is implemented using the same pipeline architecture as the inverse transform design shown in Figure 4.9. The 1-D transform units simply implement the Hadamard transform of Figure 4.6 instead of the inverse integer transform. The overall matrix multiplication and final scaling takes five clock cycles. The minimum cycle-to-cycle speed is constrained by the total delay of two 16-bit adders and twos complement operations.

4.3.3 Inverse Integer Transform

Synthesizable Digital Design

The inverse integer transform unit is shown in Figure 4.7. Input W receives 4x16-bit rows of coefficients. An integer transform is performed on each 4x4 block and a 4x16-bit column of coefficients is output through X each clock cycle.

As shown in Chapter 3, the inverse transform consists of two matrix multiplications of the input signal and the inverse transform matrices (one being the transpose of the other). Since each transform matrix is a constant, the otherwise n^2 multiplication operation can

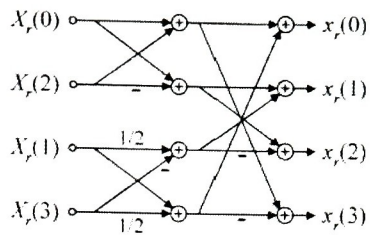

Figure 4.8: Fast Inverse Transform algorithm [12]

be reduced to a row-wise pipelined operation using the design techniques demonstrated in [18]. Shown in Figure 4.8, each row is input into a two stage butterfly structure resulting in the one-dimensional transformed row at the output. This structure is further reduced in complexity by the fact that each row value is a signed integer and the algorithm only involves shift operations and two's complement addition/subtraction.

The above structure accounts for only one row of one dimension of the overall two dimensional matrix integer transform; to utilize this efficient structure a pipelined architecture needs to be implemented. Instead of transposing the transform matrix for the second multiplication operation, the signal matrix to be transformed is transposed and sent though an identical structure as the first transform (Figure 4.8). With this methodology, the transformed output signal is itself transposed, requiring another transposition operation or the output to be taken as the columns of the desired matrix. The latter method requires no extra hardware or latency; to utilize this, downstream components must only receive the matrix as columns rather than rows.

Figure 4.9 shows the datapath of the inverse transform block. A one-dimensional transform exists on each end of the central transpose array. The transpose array consists of sixteen (4x4 matrix) register elements with three-input multiplexers acting as input selectors. The multiplexers may select from top input, side input or output feedback for a no-op cycle. Using this datapath, a matrix can be clocked in to fill the array and clocked out in the other direction to accomplish the transpose operation.

For example, consider the case when the array is empty, the direction of the register is set to down and the rows of the matrix are clocked in. After four clock cycles the array

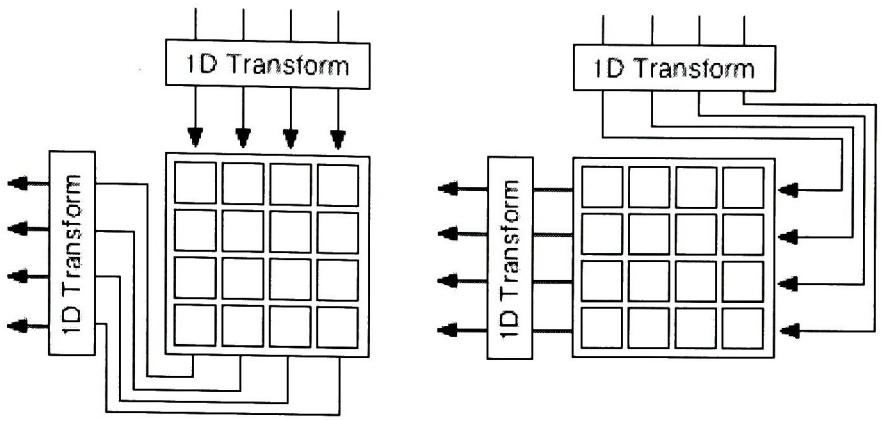

Figure 4.9: Inverse Transform datapath configurations: "down" data flow (left) and "sideways" dataflow (right)

is full with the values of the matrix after a one-dimensional transform, at which point the direction of the array is switched to sideways. The first fully transformed column of the input signal arrives at the output after the next clock cycle, after passing through the one-dimensional transform resident at the output of the transpose array. As the first column is clocked out, a new matrix to be transformed is input, filling the array in four clock cycles as the transformed array is output. After the second input matrix fills the array, the direction is changed back to down and the matrix resident in the array is clocked out in transposed form through the second inverse transform while a new matrix is clocked in. This operation continues with the direction of the transpose array being changed every four clock cycles. If for some reason the inverse transform operation needs to halt, the registers can operate in feedback mode taking their inputs from their own outputs thereby stalling the transpose operation.

The final implementation of the inverse transform block had the inputs and outputs shown in Figure 4.7. Input W is 4x16-bit wide and expects a single matrix row each clock cycle; output X is 4x8-bit wide and will output a single matrix column each cycle. The control signal (two bits wide) should be alternated between 01 and 10 every four clock cycles to control the transpose unit. A control input of 00 stalls the transform unit.

The overall matrix multiplication and final post-scale by 64 with rounding takes five clock cycles. The minimum cycle-to-cycle speed is constrained by the total delay of two 16-bit adders and twos complement operations.

4.3.4 Transform Controller

Synthesizable Digital Design

The datapath controller shown in Figure 4.10 controls the flow of data through the inverse quantization, macroblock register, and inverse transform units. Four individual state machines control the operation of each of the datapath units. The controller handles processing for an entire macroblock of coefficients, keeping track of how to transform each set of coefficients based on the macroblock type.

The controller starts the datapath processing a macroblock one clock cycle after the *mb_start* input is clocked high. As long as the *data_ready* input remains high, the controller tells the datapath to process a new set of four data coefficients each clock cycle. The controller stalls the datapath during any clock cycles when *data_ready* is low.

4.4 Intra Estimation

The intra estimation stage predicts a macroblock's samples based on previous macroblocks in the frame. Several different types of prediction are available, as described in Chapter 3.

Behavioral Model

The behavioral model of the intra estimation stage is contained in the *intra_prediction.vhdl* file. The main procedures in the model are:

intra_prediction() Checks whether the current macroblock uses Intra prediction and calls the appropriate function for the type of prediction used. Luma prediction is done using

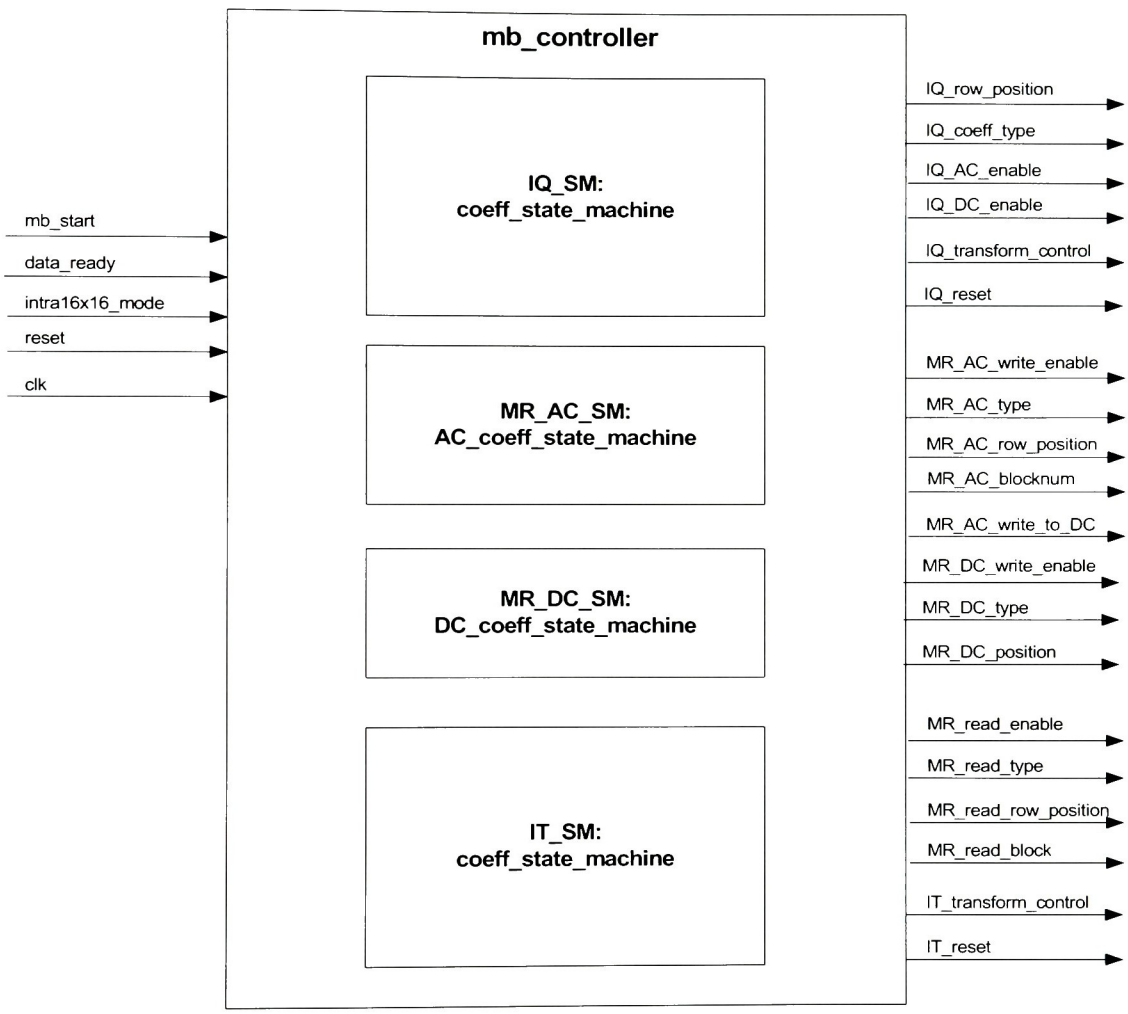

Figure 4.10: Transform controller interface for synthesizable design

either the Intra 16x16 or Intra 4x4 prediction procedures, while chroma prediction uses a separate chroma prediction procedure.

intra_4x4_prediction() Implements Intra 4x4 prediction. The type of intra estimation to use (horizontal, vertical, diagonal-down-right, etc.) is derived from the estimation types of previous macroblocks using the *derive_intra4x4_pred_mode()* procedure. The pixels in the 4x4 block immediately to the left and above the current block are then used to predict the current block's samples according to the current prediction mode.

intra_16x16_prediction() Implements Intra16x16 prediction. The type of intra estimation to use (horizontal, vertical, plane, etc.) is derived from the current macroblock type. The pixels in the 16x16 macroblock immediately to the left and above the current macroblock are then used to predict the current macroblock's samples according to the current prediction mode.

chroma_prediction() Implements Chroma prediction. This procedure is called for each of the two chroma components (Cb and Cr). The type of intra estimation to use (horizontal, vertical, diagonal-left, etc.) is derived from the *intra_chroma_pred_mode* flag. The 8x8 array of chroma pixels in the macroblock immediately to the left and above the current macroblock are then used to predict the current macroblock's samples according to the current prediction mode.

The samples were predicted using simple assignments or simple math filters based on the reference pixels from the previous macroblocks.

4.5 Inter Estimation

The inter estimation stage predicts the current macroblock's samples based on samples from a previous reference frame.

Behavioral Model

The behavioral model of the inter estimation stage is contained in the *inter_prediction.vhdl* file. The main procedures in the model are:

inter_prediction() Checks whether the current macroblock uses inter prediction. If it does, this procedure loops through each of the partitions and sub-partitions in the current macroblock. Each partition or sub-partition has its motion vector derived, then inter prediction is performed based on samples from a reference frame. The predicted samples are then stored in the current macroblock, along with motion vector information for use in predicting the motion vectors for future partitions.

derive_motion_vector() Derives the motion vector for the current partition or sub-partition. A predicted motion vector is formed based on the motion vectors of previously decoded partitions (as described in Chapter 3). The predicted motion vector is then summed with any motion vector residual data that was previously parsed from the encoded data stream to produce the final motion vector.

decode_inter_prediction() Performs inter prediction on the current partition. The correct reference frame is identified and the motion vector is used to obtain a reference set of samples from an area of the reference frame identical in size to the current partition. The reference samples are derived using interpolation if necessary because of the quarter pixel resolution of the motion vector. The current partition's samples are then derived from the reference samples, either by performing a simple assignment or by weighting the samples according to the macroblock's header parameters.

The hierarchy of the procedures used for inter prediction is shown here:

inter_prediction()
  derive_motion_vector()
    derive_neighbor_motion_data()

luma_mv_prediction()

derive_median_luma_mv_prediction()

chroma_motion_vector()

decode_inter_prediction()

ref_picture_select()

fractional_sample_interpolation()

luma_sample_interpolation()

chroma_sample_interpolation()

default_weighted_sample_prediction()

4.6 Deblocking Filter

The deblocking filter stage applies adaptive filtering to remove blocking artifacts on the edges of each 4x4 block in a macroblock. The coefficients on each block boundary are analyzed to determine the amount of filtering, if any, that should be applied to the samples on each edge of the boundary.

The deblocking filter was implemented in both a behavioral model and in synthesizable code. Only the datapath of the actual filtering was described synthesizably, however; the decision-making portion of the adaptive filter was only implemented behaviorally. This was done to allow an analysis of the dataflow without implementing the entire deblocking filter. A wrapper entity was created that contained the synthesizable datapath and the behavioral decision-making. The decisions about the filtering strength to use were then passed into the synthesizable datapath. In order to allow for easier testing, the top-levels of the behavioral model filter and the wrapped synthesizable datapath with behavioral decision-marking had identical interfaces. The deblocking filter interface is shown in Figure 4.11.

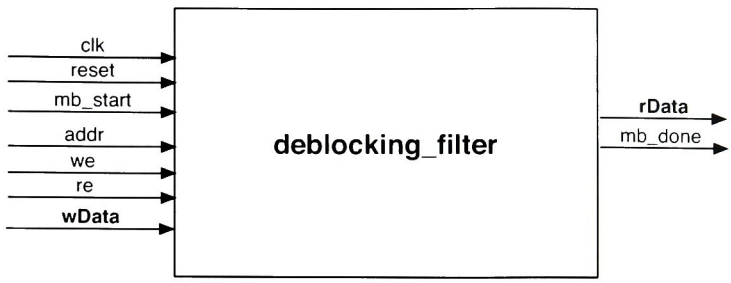

Figure 4.11: Deblocking filter interface for synthesizable design

Behavioral Model

The behavioral model of the deblocking filter is implemented in the *deblocking_filter.vhdl* file. The main procedures used in the model are:

deblocking_filter() Performs filtering on each block edge in a macroblock. This procedure uses filter_block_edges() to perform horizontal filtering of the vertical edge on the left of each column of 4x4 blocks in the macroblock. It then conducts vertical filtering of the horizontal edge on the top of each row of 4x4 blocks in the macroblock. The outputs of each filtering operation replace the macroblock samples in the frame buffer and therefore are used as inputs to future filtering operations.

filter_block_edges() Performs filtering on a single block edge (either vertical or horizontal) in a macroblock. This procedure calls filter_sample_set() with the pixels on each side of the block edge for each of the 16 luma and 8 chroma Cb or Cr lines.

filter_sample_set() Performs filtering on a line of four pixels on each side of a block edge. This procedure first derives the boundary strength and the threshold levels to use for filtering (based on the adaptive filtering algorithm described in Chapter 3). It then performs edge filtering on the current set of eight pixels.

The hierarchy of the procedures used for the deblocking filter model is shown here:

deblocking_filter_func()

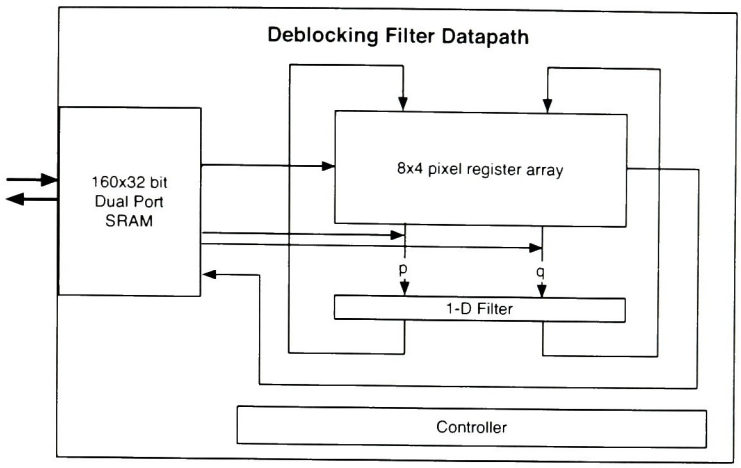

Figure 4.12: Deblocking filter datapath used in synthesizable design [11]

filter_block_edges()

filter_sample_set()

derive_boundary_strength()

derive_thresholds()

edge_filter_normal()

edge_filter_strong()

Synthesizable Digital Design

The design of the synthesizable deblocking filter datapath is based on the the design shown in Figure 4.12 [11]. The datapath consists of a dual-port memory, a register array, a reconfigurable filter, and a datapath controller.

A dual-port SRAM module is used to store the current macroblock and the bordering samples that are needed for filtering. Each 4x4 block of samples can be represented by four 32-bit data words, with each word containing four 8-bit samples. Filtering a macroblock requires 96 words to store the macroblock's luma and chroma samples, and an additional 64 words to store the bordering samples that are needed for filtering. Using a dual-port RAM component reduces the number of cycles needed for reading and writing to memory

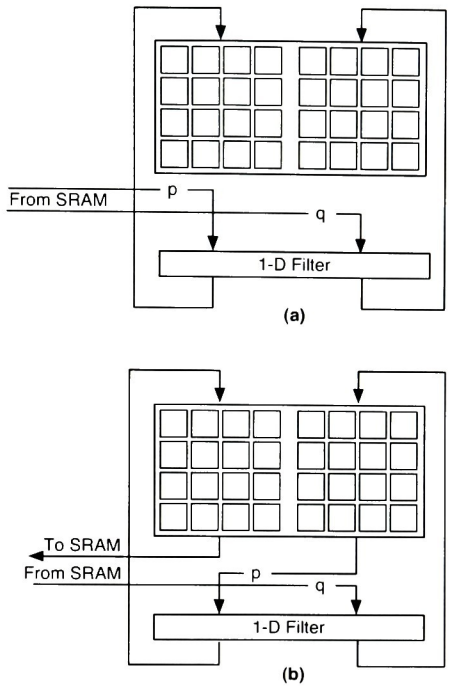

Figure 4.13: Horizontal datapath for filtering of vertical edges; (a) loading/filtering of first block (b) filtering of all other blocks [11]

and allows the block of data to be read out for processing as the previously filtered block is being written back to memory. A dual-port SRAM of 160 x 32 bits was therefore designed to meet the requirements of the datapath.

Since the samples are packed into the 32-bit wide memory in lines of four samples, the horizontal filtering path is simple: the eight samples needed for filtering can be read out of the dual-port memory and sent directly into the reconfigurable filter, taking a total of four cycles to process an entire block edge. If the processing of blocks is ordered well, one of the processed blocks can be stored in a 4x4 register array and used as one of the inputs to the next set of filtering operations. This allows the dual-port memory to be used to read out one new block for processing while simultaneously storing the filtered block of samples that is no longer needed.

The datapath for horizontal filtering is shown in Figure 4.13. Loading and filtering the first two blocks on an edge is accomplished using datapath configuration (a) to load the

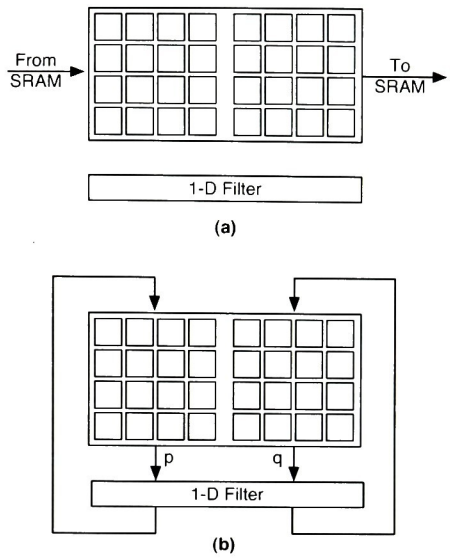

Figure 4.14: Vertical datapath for filtering of horizontal edges; (a) load/store phase (b) filtering phase [11]

two blocks simultaneously. Once the two blocks are filtered, one block is written back to memory while the other one is filtered again with the next block read from memory using datapath configuration (b).

The vertical filtering path is more complicated because a vertical line of samples cannot be read out of the memory in a single cycle. Instead, a 4x8 register array needs to be used as shown in Figure 4.14(a) to store two complete blocks of data. Once the two blocks are loaded, the datapath can be changed to send the vertical lines of samples into the reconfigurable filter as shown in Figure 4.14(b). The samples are stored back into the register array after they are filtered. After all samples have been filtered, the datapath is changed again to allow storing the two blocks of data back into memory.

The reconfigurable filter implements the sample filtering as described in Chapter 3. The filter takes the eight 8-bit pixels across a block edge and smooths any blocking artifacts found. It accepts the boundary strength and threshold values as inputs to control the amount of filtering that it applies to the current line of samples that it is processing.

A controller was also implemented to handle configuring the datapath correctly and

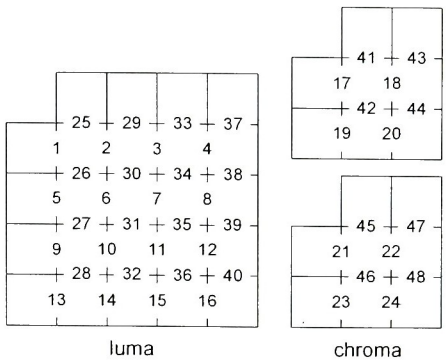

Figure 4.15: Order of edge filtering used by the controller [17]

reading and writing the correct sequence of blocks from memory. The controller state machine starts when the *mb_start* command is given the deblocking filter. The state machine assumes that all of the needed samples have been stored in the memory prior to the start command being given. Once it begins, it goes through the entire macroblock of data, reading each block, filtering each block edge, and writing the filtered samples back into memory. Filtering occurs first on vertical edges, then on horizontal edges, in the order shown in Figure 4.15. Once it has completed filtering all samples, it triggers the *mb_done* flag that is an output from the deblocking filter to indicate that the filtered data is ready for the frame buffer to read.

The deblocking filter datapath implemented here is designed to take 160 cycles to perform all horizontal filtering of vertical edges. The vertical filtering of horizontal edges takes a total of 288 cycles because of the additional data reordering that is needed through the datapath. Loading and storing each macroblock of data takes an additional 160 cycles. While designs with less latency are possible, this one was used because it provided good performance with a minimal size and a relatively uncomplicated datapath design.

Name	Description
slices	Array of slices used to decode the frame. Stores the slice header and macroblock header parameters for future reference.
LumaSamples	Two-dimensional array of the luma samples that make up the frame
ChromaSamples	Two two-dimensional arrays of the chroma Cb and Cr samples that make up the frame
ReferenceMarking	Variable storing the current reference marking of the frame
PicNum	Variable storing the short-term picture number assigned to the frame
LongTermPicNum	Variable storing the long-term picture number (if any) assigned to the frame

Table 4.3: Fields contained within the frame data structure

4.7 Frame Buffer

The frame buffer stores decoded frames for display or future reference. It is also responsible for keeping track of the current reference list and the reference markings for each frame.

Behavioral Model

The data structure used for storing decoded frames and their reference information is described in *data_types.vhdl*. The frame buffer consists of an array of frames, with each frame consisting of the fields shown in Table 4.3.

The behavioral model is implemented in *frame_buffer.vhdl*, where the following procedures are implemented:

fill_frame_macroblock() Copies a macroblock into the correct location in the frame buffer. Responsible for constructing each frame from the passed in filtered macroblock samples.

ref_picture_marking() Called after each slice is decoded. Marks the reference status of the current frame as used for short-term reference, used for long-term reference, or unused for reference.

ref_picture_list_construction() Manages the reference picture list. Called before decoding each slice to construct the current reference picture list that should be used for decoding. Looks at each of the frames in the frame buffer and constructs the reference picture list based on their reference status markings.

After the H.264/AVC decoder was implemented, the behavioral and synthesizable descriptions had to be tested. The next chapter describes how the decoder was verified.

Chapter 5

Testing

This chapter describes how verification was performed on the behavioral and synthesizable models of the H.264/AVC decoder.

5.1 Behavioral Model

Verification of the behavioral VHDL model was performed by comparing the results to the output of the reference H.264/AVC software decoder (obtained from [2]). A software reference encoder was used to compress two test video sequences. The encoded video files were then decoded using both the reference software decoder and the VHDL model. The testing environment shown in Figure 5.1 allowed an encoded video file to be used as an input to the VHDL model. Since a VHDL testbench cannot read binary data, the compressed video file was first converted to an ASCII hex representation by a C program. The text file that was generated could then be read by the VHDL testbench using the existing *hread()* procedure.

The testbench handled parsing the file NAL (Network Abstraction Layer) and sent each NAL unit to the VHDL decoder model for processing. The processed video data output from the model was then written to an output file. Again, the ASCII text file had to be processed by a C++ program to convert it into binary data. This final decoded output was then compared to the output of the reference software decoder.

Debugging the entire behavioral model on the basis of simply comparing its output to

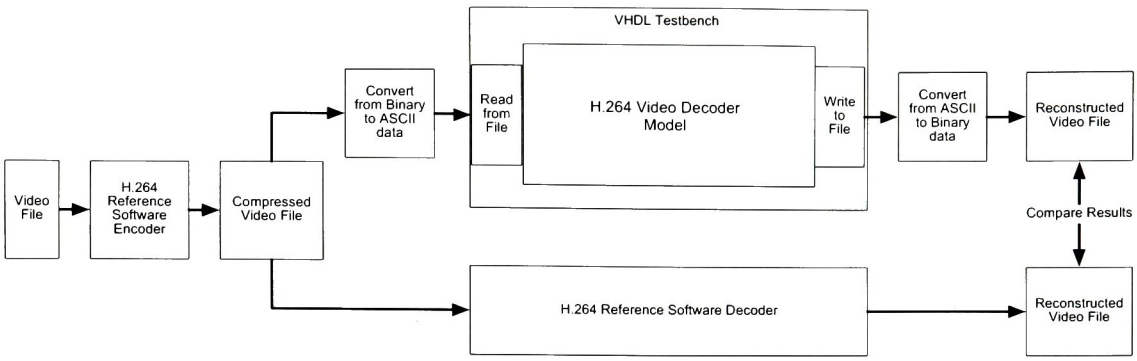

Figure 5.1: Behavioral model test setup

that of the reference software decoder would have been difficult. Although it wasn't possible to compare the outputs of individual stages against the software results, the reference software model did produce a debug text output identifying the names and values of all the data fields that were decoded from the encoded video stream. This allowed individual data fields after the run length and VLC decoders to be compared to verify that the transform and prediction stages were receiving the correct values. Because data fields have variable lengths in the type of stream encoding used by H.264/AVC, any error in the behavioral model of the run length or VLC decoders caused future parsing to occur at the wrong position in the stream. Comparing the data positions where the software parsed macroblock boundaries from the stream with the data positions used by the behavioral model allowed errors in the stream parsing to be quickly located.

Verification of the behavioral model's transform, prediction, and filtering stages had to be done based on the video image that was produced. The output video was analyzed both visually and by generating a residual comparision with the output of the software decoder. Any differences discovered had to be analyzed and identified by manually looking for bugs in the behavioral model. This process was aided by the simulator's ability to stop execution at debug breakpoints and to allow stepping through the code as it ran. Once a possible coding error was located, it was carefully compared to the algorithm described by the H.264/AVC specification. If a difference was found, the code was changed and the behavioral model testbench was ran to produce a new video output that could be analyzed

to see if any improvement was made.

This debugging and testing process was very tedious, but eventually effective in verifying that the behavioral model correctly implemented an H.264/AVC decoder. Having a successfully verified VHDL behavioral model is a powerful tool for verifying future hardware designs, since it allows the inputs and output of individual stages of the decoder to be compared. This permits each unit inside the decoder to be developed and verified independently of any of the other units.

Verification of the H.264/AVC decoder behavioral model was done using two different encoded videos, both having a frame resolution of 176x144 pixels (QCIF format). The first video consisted of a single I-frame encoded with settings that conformed to the Baseline Profile. This allowed the stream parsing, transform, and intra prediction stages to be verified without the use of inter prediction. The output video was first debugged without the deblocking filter applied. Once the output video conformed to what was visually expected, the deblocking filter was applied and the total path verified.

The second video used for verification consisted of a single I-frame followed by a single P-frame. This allowed the inter prediction unit to be verified, as well as testing the operation of the rest of the datapath on data encoded in a P-frame.

Additional verification of the behavioral model could be done using longer video sequences or by making encoded video files using different encoding parameters. However, verification of video sequences requires a large amount of memory; two QCIF frames was found to be the maximum number that could be simulated in a reasonable amount of time on a PC with 512 MB of memory and a 2.8 GHz Pentium 4 processor. The basic operation of the behavioral model decoder was successfully tested for the two types of frame encodings allowed in H.264/AVC. The two encoded video files that were used for verification tested a significant number of the possible paths through the decoder algorithm. Other test videos involving changes to minor encoding parameters would provide additionial coverage, but any potential errors found would be minor issues and easily correctable.

5.2 Synthesizable RTL Model

As discussed above, the entire video decoder was described in a behavioral model and tested successfully. In order to demonstrate how a final hardware decoder would be produced, synthesizable descriptions of the transform unit and deblocking filter modules were designed. They could then be tested against the already verified behavioral VHDL models of the individual modules to prove that each module was designed and implemented correctly. The synthesizable models for the transform unit and deblocking filter could then replace the behavioral models of those units and be simulated as part of the entire VHDL decoder.

5.2.1 Transform Unit

Testing of the transform unit was done by creating a testbench *transform_unit_comparison* entity that wrapped instantiations of both the behavioral model and the synthesizable implementation of the transform unit. The *transform_unit_comparison* entity, shown in Figure 5.2, was instantiated within the main video decoder. Each macroblock that needed to be decoded was passed into the comparison testbench, which in turn sent it into the behavioral and synthesizable transform units. The testbench waited for both units to finish processing the macroblock, then it compared the output of the synthesizable implementation with the output of the behavioral model to verify that it was identical. Any output coefficient mismatches were identified in an error message that was printed out to the simulator log.

5.2.2 Deblocking Filter

The deblocking filter was tested using a testbench *deblocking_filter_comparison* entity (shown in Figure 5.3) that instantiated both the behavioral model and the synthesizable implementation of the deblocking filter. Each macroblock that needed to be filtered was passed into the behavioral and synthesizable filters. The outputs were then compared by the testbench comparison to verify that they matched. Any output mismatches were identified by printing

an error message to the simulator log.

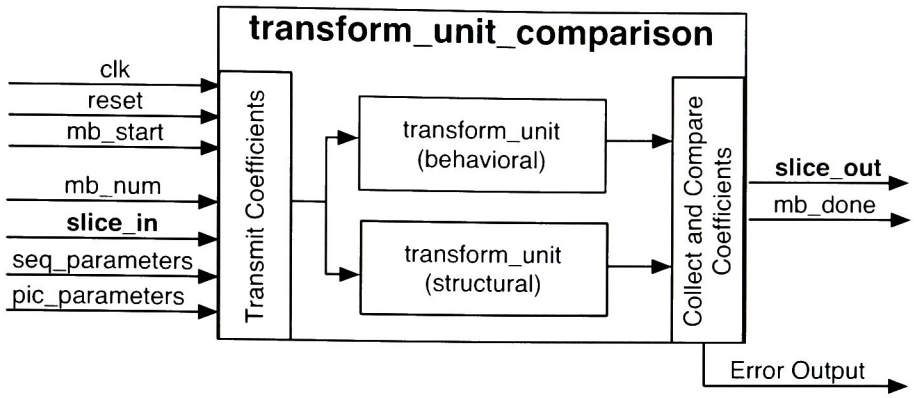

Figure 5.2: Transform unit test setup

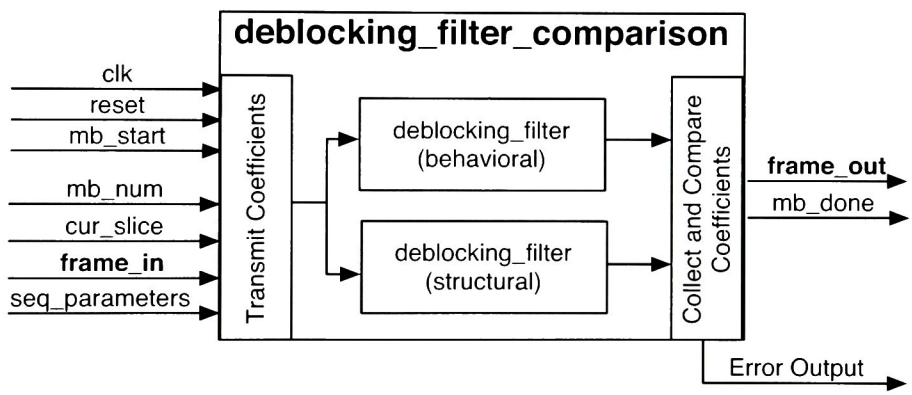

Figure 5.3: Deblocking filter test setup

Using the test procedure described in this chapter, verification was performed on the complete H.264/AVC decoder. The next chapter presents the results of testing and synthesizing the decoder implementation.

Chapter 6

Results

The H.264/AVC video decoder was tested to verify that the decoding algorithms were modeled correctly and to determine the performance of the synthesizable units. The results of running the reference software are shown and compared to the results from the behavioral model and the results of the synthesizable hardware units.

6.1 Reference Software

The reference software was used to encode and decode a sample 2-frame video sequence from the commonly used Foreman QCIF test video. The original video frames are shown in Figure 6.1.

The reference software encoder was used to encode the video sequence into an H.264/AVC

Figure 6.1: Original video sequence: Foreman frame 1 (left), Foreman frame 2 (right)

Figure 6.2: Reference software decoder output: Foreman frame 1 (left), Foreman frame 2 (right)

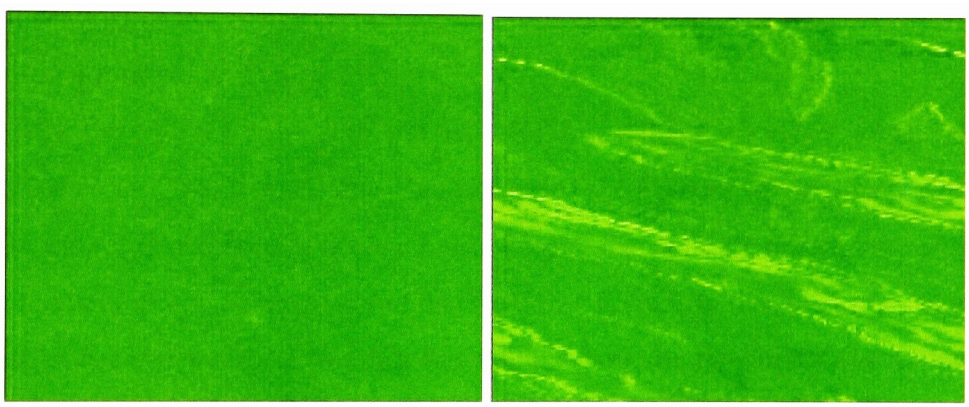

Figure 6.3: Reference software decoder difference from original video data: Foreman frame 1 (left), Foreman frame 2 (right)

Baseline Profile-compliant compressed video. The first frame of the video sequence was coded as an intra-coded I-frame that was used as a reference frame for the second frame, which was coded as a P-frame. An initial quantization parameter (QP) of 28 was used. The resulting compressed video was about 5% of the size of the original uncompressed video.

The reference software decoder was then used to decode the compressed video sequence. The video output of the decoder is shown in Figure 6.2.

A comparison of the original uncompressed video and the decoded video was generated, with the differences shown in Figure 6.3. It can be seen that the decoded video differs from the original video as a result of the lossy compression that was used. The decoded

video is still a fairly good visual representation of the original video sequence, though, as seen by comparing the decoder output in Figure 6.2 to the original video in Figure 6.1. The P-frame can be seen to have a much greater error than the I-frame.

6.2 Behavioral Model Results

The compressed video file created by the reference software encoder was used as an input to the behavioral model decoder. The resulting decoded video is shown in Figure 6.4. The output of the behavioral model decoder was compared to the output of the reference software decoder to verify that the algorithms were equivalent. The results of a difference comparison are shown in Figure 6.5.

Visually, the output frames from the behavioral model and the reference decoder look virtually identical. The second frame does appear to have a couple of jagged spots that don't appear in the reference output, however. Looking at the different comparison shows that frame 1 has very minor differences, but that frame 2 has some noticeable error. An analysis of the differences between the two decoder outputs shows that the behavioral model frame 1 output has a PSNR of 87.9 dB with a mean-squared error (MSE) of 0.000105. The frame 2 output has a PSNR of 80.26 dB with an MSE of 0.000612.

Figure 6.4: Behavioral model decoder output: Foreman frame 1 (left), Foreman frame 2 (right)

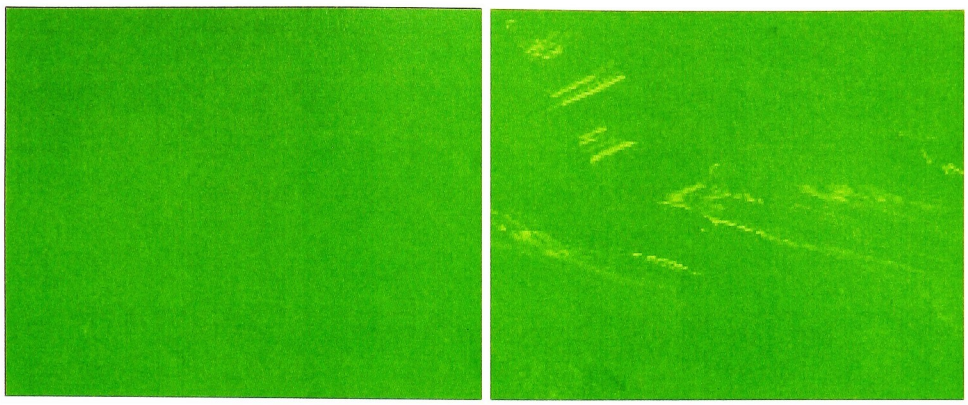

Figure 6.5: Behavioral model decoder differences from reference decoder output: Foreman frame 1 (left), Foreman frame 2 (right)

6.3 Synthesizable Hardware Results

The synthesizable hardware implementations of the transform unit and the deblocking filter were simulated to verify that they worked correctly. The comparator testbenches shown in Figures 5.2 and 5.3 were used to instantiate both the synthesizable and behavioral descriptions to compare their operation. The simulation log from the comparator testbenches showed that the outputs of the synthesizable implementations matched the outputs of the behavioral models.

Synthesis was then performed using MentorGraphics' LeonardoSpectrum tool to target two different FPGA platforms. Xilinx's Spartan-3 and Virtex-II platforms were chosen to represent both low-cost and high-performance FPGAs. Spartan-3 devices are designed for low-cost, low-power applications, while the Virtex-II devices target higher-performance use. For each platform, a 2 million gate device was chosen as being representative of an FPGA size that could be used to implement an entire decoder. The actual number of available flip-flops and function generators (FGs) varies between the two devices, however, since the Virtex-II contains a greater number of predefined multipliers and block RAM resources.

Table 6.1 shows the results of synthesizing the transform unit for the two target FPGA devices. The table shows that a clock speed of 60.5-66.0 MHz was achieved, depending on

Platform	Device	Max Clock Speed	Flip-Flops Used	FGs Used
Xilinx Spartan-3	XC3S2000	60.5 MHz	1020 (2.40%)	3531 (8.62%)
Xilinx Virtex-II	XC2V2000	66.0 MHz	1020 (4.46%)	3528 (16.41%)

Table 6.1: Synthesis results for transform unit

Platform	Device	Max Clock Speed	Flip-Flops Used	FGs Used
Xilinx Spartan-3	XC3S2000	35.7 MHz	275 (0.65%)	1469 (3.59%)
Xilinx Virtex-II	XC2V2000	42.8 MHz	275 (2.56%)	1469 (14.35%)

Table 6.2: Synthesis results for deblocking filter

the device. The transform unit requires about 1020 flip-flops and 3530 function generators for implementation on an FPGA. This was done without using any target-specific resources such as block RAMs or multipliers, however. As seen in the table, the transform unit logic used 4.46% of the available Virtex-II flip-flops and 16.41% of its available function generators, compared to only 2.40% of the flip-flops and 8.62% of the function generators in the Spartan-3.

Table 6.2 shows the results of synthesizing the deblocking filter. The synthesis results show a maximum clock rate of 35.7-42.8 MHz for the filter. The deblocking filter datapath requires about 275 flip-flops and 1469 function generators for implementation, as well as one FPGA block RAM resource for storing the data.

A performance analysis of the transform unit shows that four pixels can be processed every clock cycle, with output data having a maximum latency of 9 cycles. This means that a macroblock of data takes a total of 105 cycles to process; a complete QCIF frame of 99 macroblocks takes 10,395 cycles. Table 6.3 shows the maximum number of frames per second that can be processed on each target FPGA device, based on the maximum clock speed reported by the synthesis tool. The results show that the transform unit can easily handle processing over 30 frames per second (fps) at video resolutions all the way up to 1920x1080 (currently the highest HDTV resolution).

Video Resolution	Spartan-3	Virtex-II
CIF (320x240)	1746.0 fps	1904.8 fps
NTSC (720x480)	426.8 fps	465.5 fps
720p (1080x720)	160.1 fps	174.6 fps
1080p (1920x1080)	71.1 fps	77.6 fps

Table 6.3: Transform unit – maximum decode rate at different video resolutions

Video Resolution	Spartan-3	Virtex-II
CIF (320x240)	177.9 fps	213.3 fps
NTSC (720x480)	43.5 fps	52.1 fps
720p (1080x720)	16.3 fps	19.6 fps
1080p (1920x1080)	7.2 fps	8.7 fps

Table 6.4: Deblocking filter – maximum decode rate at different video resolutions

A performance analysis of the deblocking filter shows that a total of 608 cycles is needed to process a macroblock of video data. Table 6.4 shows the maximum number of frames per second that can be processed on each FPGA device. The results show that the current deblocking filter design can handle NTSC resolutions at 30 fps (used for DVDs), but falls short of real-time decoding for 30 fps video at 720p or 1080p resolutions.

Table 6.5 shows a comparison of the estimated power consumption and cost of the Virtex-II and Spartan-3 devices. The power consumption values are based on a complete decoder implementation using 75% of the FPGA logic and running at a clock rate of 60 MHz. The calculation was performed using a tool on Xilinx's website ([22]). The cost estimate is based on a 100+ quantity purchase using pricing available on a Xilinx distributor's

Platform	Device	Estimated Power Consumption	Approximate Cost
Xilinx Spartan-3	XC3S2000	703 mW	$61.55
Xilinx Virtex-II	XC2V2000	1586 mW	$544.50

Table 6.5: Estimated power consumption and prices for different FPGA devices

website ([5]).

Chapter 7

Conclusion

This chapter discusses the test results and performance analysis of the behavioral and synthesizable models of the H.264/AVC decoder. Based on the results that were achieved, improvements are suggested for the transform unit and deblocking filter hardware designs. A hardware architecture is also proposed for a complete implementation of an H.264/AVC video decoder.

7.1 Behavioral Model

The behavioral model was largely able to match the operation of the reference software decoder, although some differences were found between the final video outputs. Some of these differences, especially in the I-frame, can be attributed to arithmetic rounding differences. There do seem to be minor errors present in the decoding of the P-frame, however, which are most likely occurring due to incorrect decision-making logic in either the motion vector prediction or the final deblocking filter process. It is difficult to pinpoint the exact source of error, however, since the behavioral model seems to be making decisions that are consistent with the H.264/AVC standard. It may be necessary to examine the reference decoder source code running inside a debugging tool to compare its internal operation with that of the behavioral model.

Now that a complete decoder model has been developed, the testbench setup that was demonstrated can be easily used for the development of new hardware implementations of

the video decoder and its individual stages. Future work can extend the behavioral model to implement the other profiles of the H.264/AVC standard by adding support for B-frames, CABAC entropy decoding, and other features.

7.2 Synthesizable Implementation

The two synthesizable units that were implemented show how the behavioral model can be used to verify new hardware designs. The performance analysis of the synthesizable designs shows that there may be some ways to improve the transform unit and deblocking filter implementations.

Transform Unit Improvements

The transform unit design proved more than sufficient for processing even the highest HDTV resolutions in real time. The matrix multiplication datapaths required by the Hadamard and integer transforms did use a significant amount of hardware resources, however. Depending on the specific decoder application and its performance needs, it may be desirable to redesign the unit to use less hardware. One way this could be done is to reuse the 1-D transform units and increase the processing time required. Based on the current performance, doubling the time to process a macroblock could be a worthwhile design tradeoff in exchange for reducing the hardware complexity.

Deblocking Filter Improvements

The deblocking filter design proved sufficient for processing DVD-quality resolutions, but wasn't fast enough for high definition applications. Some design changes that could increase the throughput of the filtering unit are:

- Changing the order of the processing to reduce the number of memory accesses required and get rid of some of the pipeline delays neede for loading and storing data.

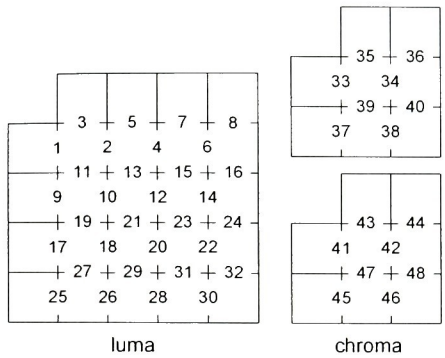

Figure 7.1: Improved deblocking filter order [17]

In the current architecture, horizontal filtering is completed before performing vertical filtering. This means each block of data is loaded from memory twice and stored back into memory twice. The throughput of the deblocking filter could be increased by enhancing the datapath to allow each block to be horizontally and vertically filtered with the blocks around it before being stored back into memory. One possible processing order, shown in Figure 7.1, has been successfully implemented in [17].

- Adding a second filtering unit to allow two filtering operations to be completed each clock cycle. This could double the throughput of the actual filtering unit.

- Adding stages to the pipeline in order to increase the maximum clock speed of the deblocking filter. The path through the actual filter is the current critical delay path; if the filtering unit were given two clock cycles to complete filtering, registers could be added to almost double the potential clock speed of this part of the circuit.

All of these design changes would require more hardware complexity, but they could significantly enhance the throughput of the deblocking filter.

7.3 Proposed Architecture for Video Decoder

The behavioral model of the video decoder was implemented without pipelining so that each stage of the decoder could be independent. A common data structure was used to hold

the video data and header information; this data was passed into each stage, processed, and the results passed to the next stage. This allowed the simulation results from each stage to be independently analyzed and verified. In an actual hardware implementation of the decoder, however, the different stages of the decoder will have to be carefully organized and pipelined in order to achieve the best throughput. Each stage will need to have access to the video data and header information that it requires, without interfering with the simultaneous operations of the other stages.

Based on the data dependencies observed in the behavioral model and the experience gained from creating two synthesizable decoder blocks, the architecture shown in Figure 7.2 is proposed for a complete hardware implementation of the video decoder. Significant changes from the generalized datapath shown previously in Figure 4.1 are described below.

Macroblock Buffers

In this architecture, the individual stages have been pipelined for simultaneous processing of macroblocks. Each pipeline stage is fed by a macroblock buffer containing the current macroblock header and data; the results of each stage are saved into another macroblock buffer for input to the next pipeline stage. In order to minimize dependencies between the pipeline stages, each macroblock buffer should be implemented as a ping-pong buffer of two macroblocks: the macroblock currently being stored by the prevous stage, and the macroblock being read by the current stage. After each macroblock has finished being processed and stored, the ping-pong bank would be swapped so the macroblock can be read by the next stage.

FPGA implementations of the decoder should make use of internal block RAM resources for the macroblock buffers. Each block RAM inside the target Xilinx FPGAs can easily provide enough dual-port RAM for a macroblock buffer.

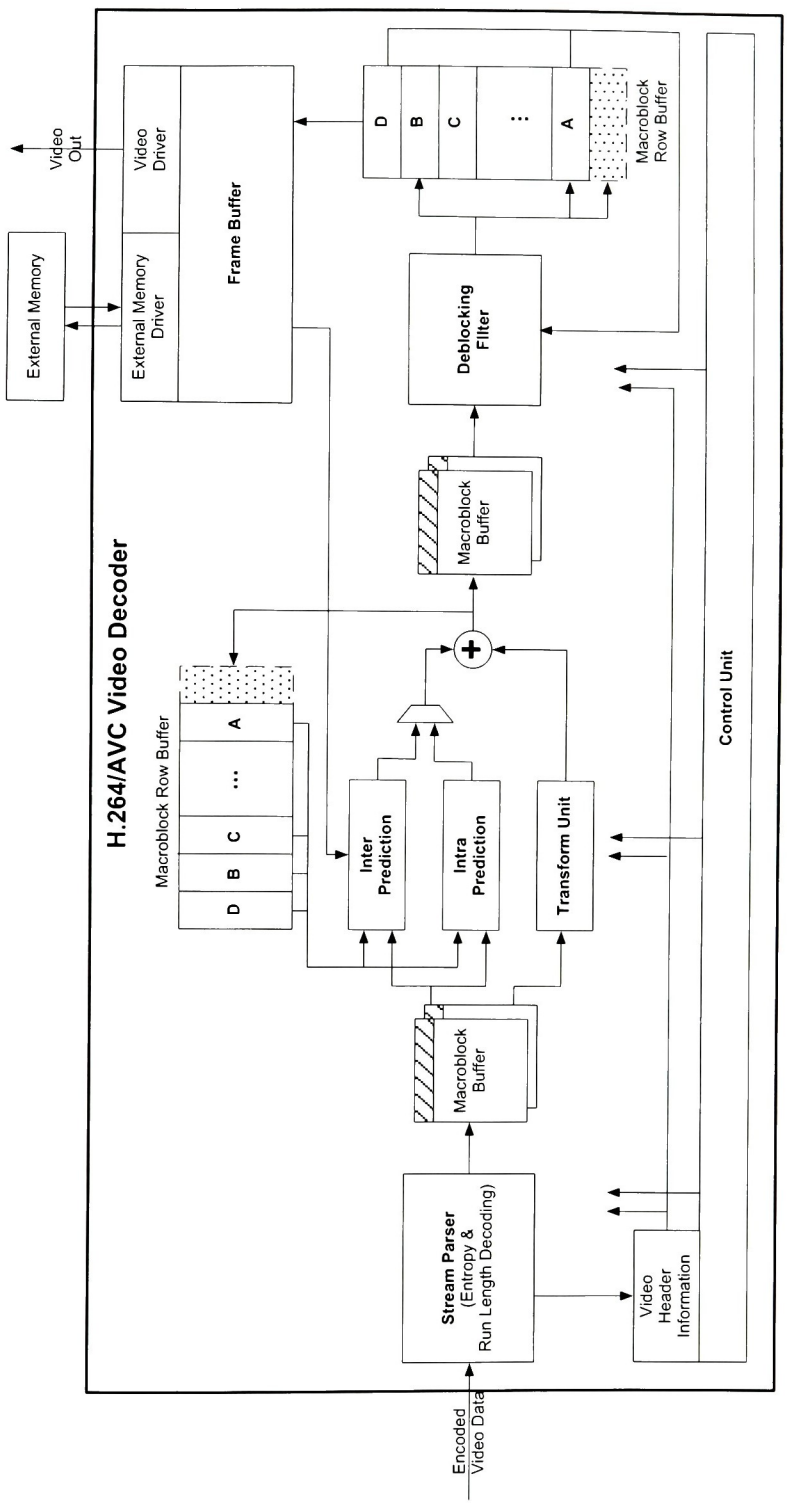

Figure 7.2: Proposed hardware architecture for H.264/AVC decoder

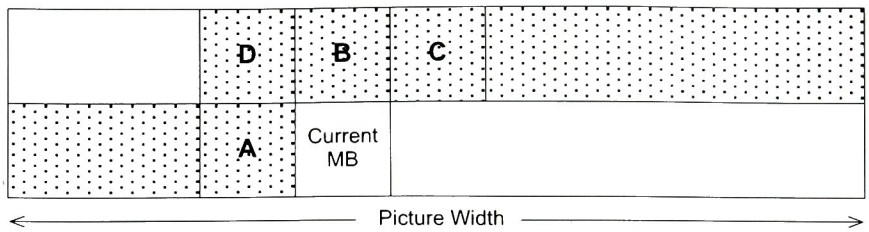

Figure 7.3: Macroblocks stored in proposed Macroblock Row Buffer

Macroblock Row Buffers

The processing performed by the intra prediction, inter prediction, and deblocking filter stages is dependent on previously decoded macroblocks that are adjacent to the current macroblock. In order to make each pipelined stage as independent as possible and reduce delays associated with memory accesses, the proposed architecture uses two buffers capable of storing a complete row of macroblocks (the required buffer size will depend on the maximum frame size that needs to be decoded). Since macroblock decoding is done in raster-scan order, the last *FrameWidthInMbs* macroblocks need to be stored to provide access to adjacent macroblocks A, B, C, and D (identified in Figure 7.3).

Two macroblock row buffers are used in the proposed architecture. One buffer contains the previous row of unfiltered macroblocks needed by the intra and inter prediction stage. The second buffer contains the previous row of filtered macroblocks needed for future filtering by the deblocking filter. The unfiltered macroblocks can be discarded when they are replaced by the next row's macroblock; the filtered macroblocks, however, represent the final decoded frame and need to be stored in the frame buffer after they are no longer needed as part of the previous row's worth of macroblocks.

External Frame Buffer Memory

After each macroblock has been decoded and is no longer needed for processing of the current frame, it is stored in the frame buffer. In the proposed architecture, the frame

Video Resolution	Minimum Needed	Maximum Needed
CIF (320x240)	0.6 MB	2.2 MB
NTSC (720x480)	2.6 MB	8.8 MB
720p (1080x720)	6.9 MB	23.5 MB
1080p (1920x1080)	15.6 MB	52.9 MB

Table 7.1: External memory required for different video resolutions

buffer controller interfaces with external memory to store decoded frames. The size requirements of the external memory depend on the maximum video resolution that will be processed. The external memory needs to be big enough to store the current frame (for display purposes) as well as the maximum number of reference frames supported. H.264/AVC specifies a maximum of 15 reference frames, although many encoded videos will require a much smaller number (*e.g.*3-5 reference frames).

An estimate of the external memory needed for various video resolutions is shown in Table 7.1. The values given show the size required for the current display frame and the current decode frame plus a minimal implementation of three reference frames or a full implementation of 15 reference frames.

Control Unit

The control unit will be responsible for controlling the flow of data through the decoder pipeline. In the proposed architecture, each of the decoding stages processes a single macroblock at a time. Each stage is given an identical amount of time to finish processing its current macroblock before the controller switches the ping-pong banks in the macroblock buffers and tells the stages to start processing the next macroblock.

The controller will be responsible for sending the appropriate video header information to each processing unit. Additionally, the control unit will need to make sure that accesses to the macroblock buffers and row buffers are managed so that each unit gets the data it needs to process the current macroblock.

Based on the complexity of the control operations needed for the video decoder, it is

suggested that the control unit be implemented using code running on a microcontroller core inside the FPGA. This will reduce the need for development of complicated hardware state machines and provide some level of flexibility for adding new video decoding capabilities.

If a microcontroller core were implemented, it seems desirable to perform the stream parsing operations in software as well. Parsing of the video header information and data parameters occurs in a well-defined, sequential order that varies depending on a large number of conditional evaluations. A pure hardware implementation of stream parsing and entropy decoding would require a extremely large amount of logic for all of the conditional evaluations required.

An attempt at moving all stream parsing and entropy decoding operations into a microcontroller core was made in [9]. Based on those results, it seems that at least some of the decoding functionality needs to remain in hardware in order to achieve real-time decoding performance. The author suggests implementing the CAVLC lookup tables in hardware, where it can be accessible by the microcontroller to speed up computations. This seems like a good partitioning scheme, especially since decoders that use CABAC instead of CAVLC will already require custom hardware to implement CABAC's complex algorithms.

The proposed hybrid hardware-software controller would therefore be partitioned so that stream parsing and general decoder control would be implemented in software running on a microcontroller, while variable-length symbol decoding (including CAVLC and CABAC) would be implemented using custom-designed hardware.

Bibliography

[1] "H.264/MPEG-4 AVC Video Compression Tutorial". http://www.lsilogic.com.

[2] Reference C++ Software. Obtained from M. Lukowiak, November 2003.

[3] Amphion. "CS6650 MPEG-2 Video Decoder Databook". http://www.amphion.com.

[4] Amphion. "MPEG-2 Video Decode in Virtex – New IP Core for a Single FPGA". http://www.amphion.com.

[5] Avnet. Avnet Electronics Marketing Home. http://www.em.avnet.com, August 2005.

[6] Y. Hu, A. Simpson, K. McAdoo, and J. Cush. "A High Definition H.264/AVC Hardware Video Decoder Core". In *IEEE International Symposium on Consumer Electronics*. Amphion Semiconductor, September 2004.

[7] Y. Huang and T. Chen. "Architecture Design for Deblocking Filter in H.264/AVC". In *Proceedings of International Conference on Image Processing*, pages 693–696, July 2003.

[8] ISO, International Organisation for Standardisation. *Text of ISO/IEC FDIS 14496-10: Information Technology – Coding of audio-visual objects – Part 10: Advanced Video Coding*, March 2003.

[9] S. Joralemon. "Hardware Software Synthesis of a H.264/AVC Baseline Profile Decoder". Master's thesis, Rochester Institute of Technology, 2005.

[10] N. Kamaci and Y. Altunbasak. "Performance comparison of the emerging H.264 video coding standard with the existing standards". *IEEE Int. Conf. Multimedia and Expo*, July 2003.

[11] P. List, A. Joch, J. Lainema, G. Bjøntegaard, and M. Karczewicz. "Adaptive deblocking filter". *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):614–619, 2003.

[12] H. S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky. "Low-complexity transform and quantization in H.264/AVC". *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):598–603, 2003.

[13] J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, and T. Wedi. "Video coding with H.264/AVC: tools, performance, and complexity". *Circuits and Systems Magazine, IEEE*, 4(1):7 – 28, May 2004.

[14] J. Ribas-Corbera, P. A. Chou, and S. L. Regunathan. A generalized hypothetical reference decoder for H.264/AVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):674–687, 2003.

[15] I. E. G. Richardson. *H.264 and MPEG-4 Video Compression.* John Wiley and Sons, Ltd., 2003.

[16] I. E. G. Richardson. "H.264/MPEG-4 Part 10 White Papers". http://www.vcodex.com/h264.html, 2004.

[17] B. Shen, W. Gao, and D. Wu. "An Implemented Architecture of Deblocking Filter for H.264/AVC". In *Proceedings of International Conference on Image Processing*, volume 1, pages 665–668, 2004.

[18] T. Wang and Y. Huang. "Parallel 4x4 2D Transform and Inverse Transform Architecture for MPEG-4 AVC/H.264". *IEEE Transactions on Circuits and Systems for Video Technology*, March 2003.

[19] T. Warsaw and W. Batts. "H.264 Inverse Transform and Quantization". Unpublished project paper for Advanced VLSI Design class, Dept. of Computer Engineering, Rochester Institute of Technology, May 2004.

[20] T. Wiegand, H. Schwarz, A. Joch, F. Kossentini, and G. J. Sullivan. "Rate-constrained coder control and comparison of video coding standards". *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):688–703, 2003.

[21] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra. "Overview of the H.264/AVC video coding standard". *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[22] Xilinx. Power Estimator Worksheet. http://www.xilinx.com, August 2005.


Memorex
Is it live or is it Memorex?™
COMPACT
disc
Recordable
www.memorex.com
CD-R
700MB 80MINUTE
52X MULTI SPEED
VHDL Modeling of an H.264 Video Decoder